

Reference Manual  
for  
**PC-lint<sup>®</sup> Plus<sup>™</sup>**  
*A diagnostic facility for C and C++*

Vector Informatik GmbH

Version 2.2

June 2024

Vector Informatik GmbH  
<https://pclintplus.com>  
<https://vector.com>

Copyright © 1985–2024 Vector Informatik GmbH  
All Rights Reserved

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without the express written permission of Vector Informatik GmbH. Digital distributions of this publication duly licensed and authorized by Vector Informatik GmbH carry implied permission for a standard computer system to perform operations necessary to store and display it (including alternative forms of presentation such as text-to-speech software used for accessibility reasons).

#### Disclaimer

Vector Informatik GmbH has taken due care in preparing this manual and the programs and data (if any) accompanying it including research, development and testing to ascertain their effectiveness.

Vector Informatik GmbH makes no warranties, express or implied, as to the contents of this manual, which is provided as is, and specifically **DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE**. Vector Informatik GmbH further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

#### Trademarks

PC-lint<sup>®</sup> is a registered trademark of Vector Informatik GmbH. PC-lint Plus<sup>™</sup> and the GS logo are trademarks of Vector Informatik GmbH. CERT<sup>®</sup> is a registered trademark of Carnegie Mellon University. MISRA<sup>®</sup> is a registered trademark of HORIBA MIRA Ltd, held on behalf of the MISRA Consortium. AUTOSAR<sup>®</sup> is a registered trademark of AUTOSAR GbR. CWE<sup>™</sup> is a trademark of The MITRE Corporation. All other products and brand names used in this manual are registered trademarks or tradenames of their respective holders.

#### Open Source Declarations

See chapter [24](#) for declarations in compliance with the licenses of Open Source Software incorporated into PC-lint Plus. These declarations may include WARRANTY DISCLAIMERS.

#### Acknowledgements

See chapter [25](#) for acknowledgements in compliance with the license for other third-party material. These acknowledgements may include WARRANTY DISCLAIMERS.

# Contents

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview	1
1.2 Flow of Execution	1
1.3 An Example	2
<b>2 Installation and Configuration</b>	<b>4</b>
2.1 System Requirements	4
2.1.1 Supported Operating Systems	4
2.1.2 Supported File Encodings	4
2.1.3 Hardware Requirements	4
2.1.4 Antivirus Software	4
2.2 Installation	4
2.2.1 Setting the PATH environment variable on Windows	5
2.2.2 Setting the PATH environment variable on Linux	5
2.2.3 Setting the PATH environment variable on macOS	5
2.3 Configuring with pclp_config	5
2.3.1 Overview	5
2.3.2 Introduction and Walkthrough of Automated Configuration with pclp_config	6
2.3.3 Creating a compiler configuration for GCC or Clang	9
2.3.4 Creating a compiler configuration for Microsoft C/C++ compilers	11
2.3.5 Creating a compiler configuration for IAR Embedded compilers	12
2.3.6 Creating a compiler configuration for Keil µVision ARMCC	13
2.3.7 Creating a compiler configuration for Microchip MPLAB X XC8/XC16/XC32	14
2.3.8 Creating a compiler configuration for MetaWare ccac ARC V2 EM/V2 HS/V3 HS/etc.	14
2.3.9 Creating a compiler configuration for Texas Instruments Code Composer Studio compilers	15
2.3.10 Creating a compiler configuration for Green Hills Software compilers	16
2.3.11 Creating a compiler configuration for SEGGER compilers	17
2.3.12 Creating a project configuration with a JSON compilation database	18
2.3.13 Creating a project configuration with make or cmake	19
2.3.14 Creating a project configuration with MSBuild on Windows	20
2.3.15 Integrating PC-lint Plus with IAR Embedded Workbench	21
2.3.16 Integrating PC-lint Plus with Keil µVision	23
2.3.17 Integrating PC-lint Plus with MPLAB X	24
2.3.18 Integrating PC-lint Plus with Eclipse-based IDEs	24
2.3.19 Integrating PC-lint Plus with Visual Studio Code	26
2.3.20 Integrating PC-lint Plus with SEGGER IDE	29
2.3.21 Using the pclpvscfg.exe GUI utility to generate a compiler and project configuration for Microsoft Visual Studio	30
2.3.22 pclp_config Options Reference	31
2.3.23 imposter Options Reference	35
2.3.24 Installing pclp_config prerequisites	37
2.3.25 Installing python	37
2.3.26 Installing required modules	37
2.4 Configuring manually	38
<b>3 The Command Line</b>	<b>42</b>
3.1 Indirect (.Int) files	42
3.2 Exit Code	42
3.3 Built-in version environment variables	42

<b>4 Options</b>	<b>44</b>
4.1 Rules for Specifying Options	44
4.1.1 Options within Comments	44
4.1.2 Lint Comments inside of Macro Definitions	44
4.1.3 Options on the Command Line	45
4.1.4 Specifying Option Arguments	45
4.1.5 Quoted Options and Arguments	45
4.1.6 Quotes in Arguments	46
4.1.7 Braces within Argument Lists	46
4.1.8 Braces and Quotes	46
4.1.9 Brace Types in Brace-Enclosed Argument Lists	47
4.1.10 Option Display	47
4.1.11 Expansion of Environment Variables in Options	47
4.1.12 Escaping Special Characters	47
4.2 Option Reference	47
4.3 Message Options	52
4.3.1 Error Inhibition	52
4.3.2 Verbosity	65
4.3.3 Message Presentation	66
4.4 Processing Options	75
4.4.1 Compiler Adaptation	75
4.4.2 Preprocessor	76
4.4.3 Tokenizing	80
4.4.4 Parsing	81
4.5 Data Options	83
4.5.1 Scalar Data Size	83
4.5.2 Scalar Data Alignment	84
4.6 Miscellaneous Options	85
4.6.1 File	85
4.6.2 Global	87
4.6.3 Output	94
4.7 Special Detection Options	95
4.7.1 Metrics	95
4.7.2 Thread Analysis	95
4.7.3 Strong Type	96
4.7.4 Miscellaneous Detection	97
4.7.5 Semantic	99
4.7.6 Value Tracking	100
4.8 Meta Characters for Options	101
4.9 How Suppression Options are Applied	102
4.10 Rules for Parameterized Suppressions	103
4.11 Flag Options	104
4.12 Compiler Adaptation	134
4.12.1 Customization Facilities	134
4.12.2 Identifier Characters	137
4.12.3 Preprocessor Statements	137
4.12.4 In-line assembly code	138
4.12.5 Pragmas	138
4.12.6 Built-in pragmas	138
4.12.7 User pragmas	139
4.12.8 Arbitrary Width Integer Types	142

<b>5</b>	<b>Libraries</b>	<b>143</b>
5.1	Library Header Files	143
5.2	Library Modules	146
5.3	Assembly Language Modules	146
<b>6</b>	<b>Precompiled Headers</b>	<b>148</b>
6.1	Introduction to precompiled headers	148
6.2	Designating the precompiled header	149
6.3	Monitoring precompiled headers	149
6.4	The use of make files	149
6.5	Lint comment options and precompiled headers	150
<b>7</b>	<b>Strong Types</b>	<b>151</b>
7.1	Rationale	151
7.2	Creating Strong Types with <code>-strong</code>	151
7.3	Strong Types for Array Indices	152
7.4	Dimensional Analysis	154
7.4.1	Dimensional Types	155
7.4.2	Dimensionally Neutral Types	155
7.4.3	Antidimensional Types	155
7.4.4	Multiplication and Division of Dimensional Types	156
7.4.5	Dimensional Types and the <code>%</code> operator	157
7.4.6	Conversions	157
7.4.7	Integers	158
7.5	Strong Type Hierarchies	159
7.5.1	The Need for a Type Hierarchy	159
7.5.2	The Natural Type Hierarchy	160
7.5.3	Adding to the Natural Hierarchy	161
7.5.4	Restricting Down Assignments ( <code>-father</code> )	162
7.6	Printing the Hierarchy Tree	163
7.7	Reference Information	163
7.7.1	Full Source for the Gravitation Example	163
7.7.2	The Strong Type of an Expression	164
7.7.3	Canonical Form for Dimensional Strong Types	164
7.7.4	Message Numbers	164
<b>8</b>	<b>Value Tracking</b>	<b>165</b>
8.1	Introduction	165
8.1.1	Anatomy of a Value Tracking Message	165
8.2	Value Inferencing	166
8.2.1	Conditionals	166
8.2.2	Assertions	166
8.3	Integer Range Tracking	166
8.4	Terminology	167
8.5	Value Display Format	167
8.5.1	Integers	167
8.5.2	Pointers in General	167
8.5.3	Pointers to a Single Datum	167
8.5.4	Pointers to Buffers with Multiple Elements	167
8.5.5	Objects of Structure or Class Type	168
8.5.6	File Streams	168
8.5.7	Uninitialized Values	168
8.6	General Usage	169
8.7	Debugger	169
8.8	Interfunction and Intermodule Value Tracking	170

8.9	Limitations	170
8.9.1	Initial Values of Static Variables	170
8.9.2	The Correlated Variables Problem	170
8.9.3	Terminal Depth Assistance	171
8.10	Changes from Older Products	171
<b>9</b>	<b>Semantics</b>	<b>172</b>
9.1	Function Mimicry (-function)	172
9.1.1	Special Functions	172
9.1.2	Function Listing	174
9.1.3	Other names with special behavior that cannot be mimicked	193
9.2	Semantic Specifications (-sem)	193
9.2.1	Possible Semantics	194
9.2.2	Semantic Expressions	201
9.2.2.1	Return Semantics	201
9.2.2.2	Return Semantic Validation	203
9.2.2.3	Function-wide semantics	204
9.2.2.4	Overload-Specific Semantics	205
9.2.3	Notes on Semantic Specifications	207
<b>10</b>	<b>Metrics</b>	<b>209</b>
10.1	Introduction	209
10.1.1	Terminology	209
10.1.2	Options	209
10.2	Metric Report	209
10.2.1	Metric Nomination	210
10.2.2	Report Fields	210
10.3	Metric Expressions	210
10.3.1	Built-in Functions	210
10.4	Metric Checking	211
10.4.1	Metric Violation Messages	211
10.4.2	Integration with Queries	212
10.5	Custom Metrics	213
10.5.1	Examples	213
10.5.1.1	Percentage of functions in a file with multiple <b>return</b> statements	213
10.5.1.2	Number of inheritance edges in class hierarchy	214
10.5.1.3	Ratio of number of lines in member function to average number of lines in member functions of the containing class	214
10.6	Built-in Metrics	214
10.6.1	Definitions	214
10.6.2	Project ( <b>project</b> )	215
10.6.3	Translation Unit ( <b>translation_unit</b> )	217
10.6.4	File ( <b>file</b> )	217
10.6.5	Function ( <b>function</b> )	218
10.6.6	Class ( <b>class</b> )	223
10.6.7	Dynamic Metrics	226
10.7	Sample Derived Metrics	226
10.7.1	Class ( <b>class</b> )	226
<b>11</b>	<b>Thread Analysis</b>	<b>227</b>
11.1	Introduction	227
11.2	Library Support	227
11.3	Identifying Threads	227
11.3.1	Options to Identify Threads	227
11.3.2	Automatic Identification of Threads	229

11.4	Mutual Exclusion . . . . .	229
11.4.1	Trylock-like Functions . . . . .	231
11.4.2	Locker Classes . . . . .	232
11.4.3	Shared Locks . . . . .	233
11.4.4	Internal Global Recursive Mutex . . . . .	233
11.5	Function Pointers . . . . .	233
11.6	Thread Unfriendly Functions . . . . .	234
11.6.1	Category 1 Functions . . . . .	234
11.6.2	Category 2 Functions . . . . .	235
11.6.3	Category 3 Function . . . . .	236
11.6.4	Category 4 Function . . . . .	236
11.6.5	Category 5 Function . . . . .	237
11.7	Thread Local Storage . . . . .	237
11.7.1	__Thread_local . . . . .	237
11.7.2	thread_local . . . . .	237
11.7.3	__thread . . . . .	237
11.7.4	__declspec(thread) . . . . .	238
11.8	Limitations . . . . .	238
11.9	Thread Analysis Reports . . . . .	238
11.9.1	Field Types and Representation . . . . .	239
11.9.2	Thread Reports . . . . .	239
11.9.3	Function Reports . . . . .	239
11.9.4	Mutex Reports . . . . .	240
11.9.5	Variable Reports . . . . .	241
11.10	Message Summary . . . . .	241
11.11	Thread Analysis Phases . . . . .	242
11.11.1	Inhibition of Thread Analysis . . . . .	242
11.12	Supporting Other Thread Libraries . . . . .	242
11.12.1	Thread Semantics . . . . .	243
11.12.2	Mutex Semantics . . . . .	244
11.12.3	Locker Semantics . . . . .	247
11.12.4	Trylock Semantics . . . . .	248
<b>12</b>	<b>MISRA<sup>®</sup> Standards Checking</b>	<b>250</b>
12.1	MISRA C 2012 . . . . .	252
12.1.1	MISRA C 2012 Guideline Support Summary . . . . .	252
12.1.2	MISRA C 2012 Guideline Support Matrix . . . . .	252
12.2	MISRA C 2012 AMD-1 . . . . .	256
12.2.1	MISRA C 2012 AMD-1 Guideline Support Summary . . . . .	256
12.2.2	MISRA C 2012 AMD-1 Guideline Support Matrix . . . . .	256
12.3	MISRA C 2012 AMD-2 . . . . .	257
12.3.1	MISRA C 2012 AMD-2 Guideline Support Summary . . . . .	257
12.3.2	MISRA C 2012 AMD-2 Guideline Support Matrix . . . . .	257
12.4	MISRA C 2012 AMD-3 . . . . .	258
12.4.1	MISRA C 2012 AMD-3 Guideline Support Summary . . . . .	258
12.4.2	MISRA C 2012 AMD-3 Guideline Support Matrix . . . . .	258
12.5	MISRA C 2012 AMD-4 . . . . .	259
12.5.1	MISRA C 2012 AMD-4 Guideline Support Summary . . . . .	259
12.5.2	MISRA C 2012 AMD-4 Guideline Support Matrix . . . . .	259
12.6	MISRA C++ . . . . .	260
12.6.1	MISRA C++ Guideline Support Summary . . . . .	260
12.6.2	MISRA C++ Guideline Support Matrix . . . . .	260
12.7	MISRA C 2004 . . . . .	266
12.7.1	MISRA C 2004 Guideline Support Summary . . . . .	266

12.7.2 MISRA C 2004 Guideline Support Matrix . . . . .	266
<b>13 AUTOSAR<sup>®</sup> Standard Checking</b>	<b>270</b>
13.1 Introduction to AUTOSAR Support . . . . .	270
13.2 AUTOSAR19 . . . . .	272
13.2.1 AUTOSAR19 Guideline Support Summary . . . . .	272
13.2.2 AUTOSAR19 Guideline Support Matrix . . . . .	272
13.3 AUTOSAR17 . . . . .	281
13.3.1 AUTOSAR17 Guideline Support Summary . . . . .	281
13.3.2 AUTOSAR17 Guideline Support Matrix . . . . .	281
<b>14 CERT<sup>®</sup> C Standard Checking</b>	<b>289</b>
14.1 Introduction to CERT C Support . . . . .	289
14.2 CERT C Support . . . . .	291
14.2.1 CERT C Guideline Support Summary . . . . .	291
14.2.2 CERT C Guideline Support Matrix . . . . .	291
<b>15 CWE<sup>™</sup> Checking</b>	<b>303</b>
15.1 Introduction to CWE Support . . . . .	303
15.2 CWE Support . . . . .	304
15.2.1 CWE Support Summary . . . . .	304
15.2.2 CWE Support Matrix . . . . .	304
<b>16 Queries</b>	<b>311</b>
16.1 Introduction . . . . .	311
16.2 Overview . . . . .	311
16.3 Query Fundamentals . . . . .	312
16.3.1 Types and Values . . . . .	312
16.3.2 Query Execution . . . . .	312
16.3.3 Numeric Operators . . . . .	313
16.3.3.1 Multiplicative and Additive Operators . . . . .	313
16.3.3.2 Bitwise Operators . . . . .	313
16.3.4 String Operators . . . . .	313
16.3.4.1 Textual Portrayal of Types . . . . .	313
16.3.5 The AST . . . . .	314
16.3.5.1 Conversion Functions . . . . .	314
16.3.5.2 The <code>currentnode</code> Operator . . . . .	316
16.3.5.3 The <i>with-node</i> ( <code>:</code> ) Operator . . . . .	316
16.3.6 Relational Operators . . . . .	316
16.3.7 Variables and Assignment . . . . .	317
16.3.8 The <code>message</code> Operator . . . . .	317
16.3.9 The <code>verbosify</code> Operator . . . . .	318
16.3.10 Logical Operators . . . . .	318
16.3.10.1 Alternate Spellings . . . . .	318
16.3.11 The <code>if</code> Expression . . . . .	319
16.3.11.1 The Type and Value of an <code>if</code> Expression . . . . .	319
16.3.12 The <code>while</code> Expression . . . . .	319
16.3.13 The <code>echo</code> Operator . . . . .	319
16.3.14 Compound Expressions . . . . .	321
16.3.15 Parentheses . . . . .	322
16.3.16 Operator Precedence . . . . .	322
16.4 Builtin Functions . . . . .	322
16.4.1 Function Call Syntax . . . . .	322
16.4.2 Default Arguments . . . . .	323
16.4.3 Naming Conventions . . . . .	323



16.5	Examples	323
16.5.1	-astquery Examples	323
16.5.2	-eqquery Examples	327
16.6	Regular Expression Basics	330
16.6.1	Regular Characters and Metacharacters	330
16.6.2	Pattern Anchoring	331
16.6.3	Dot and Repetition	331
16.6.4	Alternation	331
16.6.5	Subpatterns	331
16.6.6	Backslash	331
16.6.7	Character Classes	332
16.6.8	Captures	333
16.6.9	Mode Modifiers	333
16.7	Debugging Queries	333
16.7.1	The <b>assert</b> Operator	333
16.7.2	Dumping the Query Tree	334
16.8	Builtin Function List	335
16.8.1	Non-member functions	335
16.8.2	ASTNode functions	336
16.8.3	ASTType functions	343
16.8.4	ArraySubscriptExpr functions	347
16.8.5	AttributedStmt functions	348
16.8.6	BinaryOperator functions	348
16.8.7	CXXBoolLiteralExpr functions	348
16.8.8	CXXCatchStmt functions	348
16.8.9	CXXConstructExpr functions	349
16.8.10	CXXConstructorDecl functions	349
16.8.11	CXXConversionDecl functions	349
16.8.12	CXXDestructorDecl functions	350
16.8.13	CXXForRangeStmt functions	350
16.8.14	CXXMethodDecl functions	350
16.8.15	CXXRecordDecl functions	351
16.8.16	CXXTryStmt functions	354
16.8.17	CallExpr functions	354
16.8.18	CastExpr functions	355
16.8.19	CharacterLiteral functions	355
16.8.20	CompoundStmt functions	355
16.8.21	CoreturnStmt functions	355
16.8.22	CoroutineBodyStmt functions	356
16.8.23	Decl functions	356
16.8.24	DeclRefExpr functions	357
16.8.25	DoStmt functions	357
16.8.26	EnumConstantDecl functions	357
16.8.27	EnumDecl functions	357
16.8.28	Expr functions	358
16.8.29	FieldDecl functions	359
16.8.30	FloatingLiteral functions	360
16.8.31	ForStmt functions	360
16.8.32	FriendDecl functions	360
16.8.33	FriendTemplateDecl functions	360
16.8.34	FunctionDecl functions	360
16.8.35	GotoStmt functions	363
16.8.36	IfStmt functions	363
16.8.37	IntegerLiteral functions	363

16.8.38 LabelDecl functions	363
16.8.39 LabelStmt functions	363
16.8.40 LambdaExpr functions	363
16.8.41 LinkageSpecDecl functions	364
16.8.42 Location functions	364
16.8.43 MemberExpr functions	364
16.8.44 NamedDecl functions	365
16.8.45 NamespaceAliasDecl functions	365
16.8.46 NamespaceDecl functions	365
16.8.47 NullStmt functions	365
16.8.48 ParenExpr functions	365
16.8.49 ParmVarDecl functions	366
16.8.50 RecordDecl functions	366
16.8.51 ReturnStmt functions	366
16.8.52 StaticAssertDecl functions	366
16.8.53 Stmt functions	367
16.8.54 String functions	367
16.8.55 SwitchCase functions	367
16.8.56 SwitchStmt functions	367
16.8.57 TagDecl functions	368
16.8.58 TypeDecl functions	368
16.8.59 TypedefNameDecl functions	368
16.8.60 UnaryOperator functions	368
16.8.61 UsingDecl functions	369
16.8.62 UsingDirectiveDecl functions	369
16.8.63 ValueDecl functions	369
16.8.64 ValueStmt functions	369
16.8.65 VarDecl functions	369
16.8.66 WhileStmt functions	370
<b>17 Other Features</b>	<b>371</b>
17.1 Format Checking	371
17.1.1 Dangerous Use	371
17.1.2 Argument Inconsistencies	372
17.1.3 Positional Arguments	372
17.1.4 Non-ISO features	372
17.1.5 Incorrect Format Specifiers	373
17.1.6 Suspicious Format Specifiers	373
17.1.7 Elective Notes and Customization	373
17.2 Precision, Viable Bit Patterns, and Representable Values	374
17.3 Static Initialization	375
17.4 Indentation Checking	375
17.5 Size of Scalars	377
17.6 Stack Usage Report	378
17.7 Migrating to 64 bits	380
17.8 Deprecation of Entities	380
17.8.1 Deprecation of Options	381
17.8.2 Deprecation of Types	381
17.8.3 Deprecation of Preprocessor Directives	382
17.8.4 Deprecation of Format Function Conversion Specifiers	382
17.9 Parallel Analysis	382
17.10 Language Limits	383

<b>18 Preprocessor</b>	<b>386</b>
18.1 Preprocessor Symbols	386
18.2 #include Processing	387
18.2.1 INCLUDE Environment Variable	387
18.3 ANSI/ISO Preprocessor Facilities	388
18.3.1 #line and #	388
18.4 Non-Standard Preprocessing	388
18.4.1 #import	388
18.4.2 #include_next	389
18.4.3 #ident	389
18.4.4 #sccs	389
18.4.5 #warning	389
18.5 User-Defined Keywords	389
18.6 Preprocessor sizeof	389
<b>19 Living with Lint</b>	<b>391</b>
19.1 An Example of a Policy	391
19.2 Recommended Setup	392
19.3 Final Thoughts	393
<b>20 Common Issues</b>	<b>394</b>
20.1 Syntax errors when using Boost	394
20.2 Suppressing MISRA/AUTOSAR/CERT C/CWE messages from library/system headers	394
20.3 Standard C++ keywords not recognized when using <code>au-misra-cpp.lnt</code>	394
20.4 Running <code>pclp_config</code> from Windows produces errors or no result	394
20.5 Resolving error 309 ( <code>#error</code> directive encountered)	394
20.6 Resolving error 330 (static assert failed)	395
20.7 Resolving errors 305 (unable to open module) and 307 (unable to open indirect file)	395
20.8 Syntax errors when using options that contain spaces	396
20.9 Resolving error 322 (unable to open include file)	396
20.10 Syntax errors when using options containing parentheses on the command prompt	397
<b>21 Frequently Asked Questions</b>	<b>398</b>
21.1 How do I keep PC-lint Plus from processing header files?	398
21.2 When do I need to use <code>imposter</code> ?	398
21.3 How should <code>imposter</code> be compiled?	398
21.4 What can I do to improve analysis speed?	398
21.5 PC-lint Plus has crashed, what do I do next?	398
21.6 Why isn't my option working?	399
21.7 How can I temporarily disable suppressions of one or more messages?	399
21.8 How can I customize existing messages or add new messages?	399
21.9 How can I change the message category associated with a message?	399
21.10 How can I generate XML/HTML output?	400
21.11 Can PC-lint Plus generate custom reports?	400
21.12 Where can I find <code>msg.txt</code> / <code>msg.json</code> ?	400
21.13 How can I apply options only to C files but not C++ files (or vice versa)?	400
21.14 Can PC-lint Plus recursively scan sub directories for the source code?	400
<b>22 Messages</b>	<b>401</b>
22.1 Message Parameters	403
22.2 Messages 1-999	403
22.2.1 Messages 1-99	403
22.2.2 Messages 100-199	409
22.2.3 Messages 300-399	412
22.2.4 Messages 400-499	415

22.2.5	Messages 500-599	434
22.2.6	Messages 600-699	451
22.2.7	Messages 700-799	470
22.2.8	Messages 800-899	486
22.2.9	Messages 900-999	499
22.3	Messages 1000-1999	512
22.3.1	Messages 1000-1099	512
22.3.2	Messages 1100-1199	519
22.3.3	Messages 1300-1399	522
22.3.4	Messages 1400-1499	522
22.3.5	Messages 1500-1599	525
22.3.6	Messages 1700-1799	535
22.3.7	Messages 1900-1999	549
22.4	Messages 2000-2999	556
22.4.1	Messages 2000-2099	556
22.4.2	Messages 2400-2499	556
22.4.3	Messages 2500-2599	573
22.4.4	Messages 2600-2699	575
22.4.5	Messages 2700-2799	578
22.4.6	Messages 2800-2899	582
22.4.7	Messages 2900-2999	582
22.5	Messages 3000-3999	584
22.5.1	Messages 3400-3499	584
22.5.2	Messages 3700-3799	590
22.5.3	Messages 3900-3999	592
22.6	Messages 4000-6999	592
22.7	Messages 8000-8999	593
22.7.1	Messages 8000-8099	593
22.8	Messages 9000-9999	593
22.8.1	Messages 9000-9099	593
22.8.2	Messages 9100-9199	610
22.8.3	Messages 9200-9299	622
22.8.4	Messages 9400-9499	629
22.8.5	Messages 9500-9599	636
22.8.6	Messages 9900-9999	638
<b>23</b>	<b>Revision History</b>	<b>640</b>
23.1	Version 2.2	640
23.1.1	Highlights	640
23.1.2	Summary	640
23.1.2.1	New Features	640
23.1.2.2	Improvements	640
23.1.2.3	Bugs Fixed	640
23.1.2.4	Known Issues	641
23.1.3	New Features	641
23.1.4	Improvements	641
23.1.5	Bugs Fixed	642
23.1.6	Known Issues	644
23.2	Version 2.1	646
23.2.1	Highlights	646
23.2.2	Summary	646
23.2.2.1	New Features	646
23.2.2.2	Improvements	646
23.2.2.3	Bugs Fixed	648

23.2.2.4	Known Issues	649
23.2.2.5	AUTOSAR Summary	649
23.2.2.6	MISRA Summary	649
23.2.3	New Features	650
23.2.4	Improvements	651
23.2.5	Bugs Fixed	660
23.2.6	Known Issues	664
23.3	Version 2.0	666
23.3.1	Highlights	666
23.3.2	Summary	666
23.3.2.1	New Features	666
23.3.2.2	Improvements	666
23.3.2.3	Bugs Fixed	668
23.3.2.4	AUTOSAR Summary	668
23.3.2.5	CERT C Summary	669
23.3.2.6	MISRA Summary	669
23.3.3	New Features	669
23.3.4	Improvements	670
23.3.5	Bugs Fixed	678
23.3.6	Known Issues	680
23.3.7	Changes to Beta Functionality Since Version 2.0 Beta 3	682
23.4	Version 1.4.1	683
23.4.1	Summary	683
23.4.1.1	Bugs Fixed	683
23.5	Version 1.4	683
23.5.1	Summary	683
23.5.1.1	New Features	683
23.5.1.2	Improvements	683
23.5.1.3	Bugs Fixed	685
23.5.1.4	Known Issues	686
23.5.1.5	AUTOSAR Summary	686
23.5.1.6	CERT C Summary	686
23.5.1.7	MISRA Summary	687
23.6	Version 1.3.5	687
23.6.1	Summary	687
23.6.1.1	Improvements	687
23.6.1.2	Bugs Fixed	688
23.6.1.3	Known Issues	689
23.6.1.4	AUTOSAR	690
23.6.1.5	MISRA C 2012	690
23.6.1.6	MISRA C 2004	690
23.6.1.7	MISRA C++	690
23.7	Version 1.3	690
23.7.1	Summary	690
23.7.1.1	Bugs Fixed	690
23.7.1.2	Documentation Improvements	691
23.7.1.3	General Improvements	691
23.7.1.4	MISRA C 2012 Improvements	693
23.7.1.5	MISRA C++ Improvements	693
23.7.1.6	New Features	693
23.8	Version 1.2	695
23.8.1	Summary	695
23.8.1.1	Bugs Fixed	695
23.8.1.2	MISRA C 2012 Improvements	695

23.8.1.3	MISRA C++ Improvements	695
23.8.1.4	General Improvements	696
23.8.1.5	New Features	696
23.8.1.6	Documentation Enhancements	697
23.9	Version 1.1	697
23.9.1	Summary	697
23.9.1.1	Bugs Fixed	697
23.9.1.2	MISRA C 2004 Improvements	698
23.9.1.3	MISRA C 2012 Improvements	698
23.9.1.4	MISRA C++ Improvements	698
23.9.1.5	General Improvements	699
23.9.1.6	New Features	699
23.9.1.7	Documentation Enhancements	700
23.10	Differences from PC-lint® 9.0	700
23.10.1	Major New Features	701
23.10.2	General Diagnostic Changes	702
23.10.3	Value Tracking	703
23.10.4	Semantics	704
23.10.5	Strong Types and Dimensional Analysis	706
23.10.6	Improved Format String Checking	706
23.10.7	Miscellaneous enhancements and Quiet Changes	706
23.10.8	Error Inhibition	707
23.10.9	Verbosity	707
23.10.10	Message Presentation	707
23.10.11	Miscellaneous Options	707
23.10.11.1	Global	707
23.10.11.2	Output	707
23.10.11.3	New Flag Options	707
<b>24</b>	<b>Open Source Declarations</b>	<b>709</b>
<b>25</b>	<b>Acknowledgements</b>	<b>712</b>
25.1	Carnegie Mellon University	712
25.2	MITRE Corporation	712
<b>26</b>	<b>Bibliography</b>	<b>714</b>

# 1 Introduction

## 1.1 Overview

PC-lint Plus is a static analysis tool that finds bugs, quirks, idiosyncrasies, and glitches in C and C++ programs. The purpose of this analysis is to determine potential problems in such programs before integration or porting, or to reveal unusual constructs that may be a source of subtle and, yet, undetected errors. Because it looks across several modules rather than just one, it can determine things that a compiler cannot. It is normally much fussier about many details than a compiler wants to be.

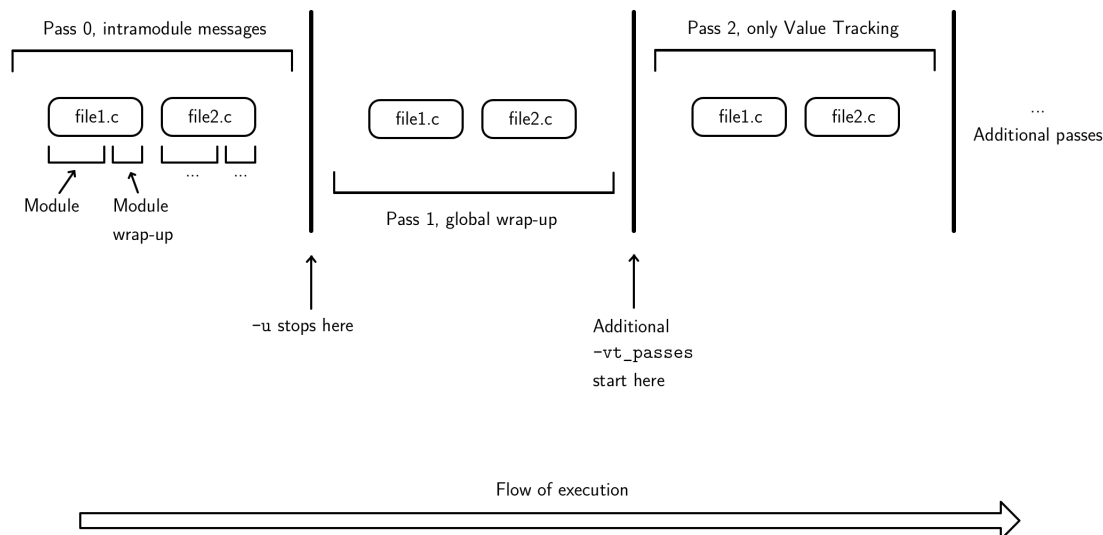
PC-lint Plus doesn't just find bugs and potential bugs, such as null pointer dereference, out of bounds access, and improper operation order, it will also point out the use of dubious constructs and substandard practices that are likely to result in buggy code or code that is difficult to reason about. PC-lint Plus can also be configured to diagnose violations of common coding standard violations, such as the MISRA C and MISRA C++ guidelines. Each diagnostic message is assigned a unique number and a great deal of flexibility is provided for the control of when and how messages are delivered. In particular, the format of the messages emitted is extremely customizable and messages can be enabled/disabled for individual files, functions, lines and for specific symbols, calls, expressions, statements, types, etc.

PC-lint Plus also offers a number of distinguishing flagship features including:

- Value Tracking
- Strong Type Checking
- Dimensional Analysis
- User-defined Function Semantics

See the corresponding sections of this reference manual for additional information.

## 1.2 Flow of Execution



By default, PC-lint Plus performs 2 passes over the source code. In each pass, a new thread is launched to process each file in your project. The `-max_threads` option controls the number of *concurrent* processing threads. Even if only one concurrent processing thread is requested, the processing of each file will take place in an independent thread.

In pass 0, most standard messages that do not require aggregate information about entire files or the entire project are emitted as PC-lint Plus examines the code. At the end of each module in pass 0, module wrap-up

is performed and messages that rely on information gathered from the entirety of the module are issued (e.g. reports of unused entities not visible outside the module).

In pass 1, information gathered from all the modules in pass 0 is used to perform global wrap-up. During global wrap-up, messages that rely on this inter-module information are issued (e.g. reports of unused entities with external linkage). Value Tracking is also performed for calls between functions in different modules collected during intramodule Value Tracking during pass 0.

The `-unit_check` option stops execution before pass 1, effectively suppressing global wrap-up. This option is often used when analyzing a subset of a larger project. The `-vt_passes` option can be used to request additional Value Tracking passes after pass 1. The benefits of multiple Value Tracking passes is discussed in section [8.8 Interfunction Value Tracking](#).

### 1.3 An Example

Consider the following C program (we have deliberately kept this example small and comprehensible):

```

1  char *report(short m, short n, char *p) {
2      int result;
3      char *temp;
4      long nm;
5      int i, k, kk;
6      char name[11] = "Joe Jakeson";
7      nm = n * m;
8      temp = p == "" ? "null" : p;
9      for (i = 0; i < m; i++) {
10         k++;
11         kk = i;
12     }
13     if (k == 1)
14         result = nm;
15     else if (kk > 0)
16         result = 1;
17     else if (kk < 0)
18         result = -1;
19     if (m == result)
20         return temp;
21     else
22         return name;
23 }
```

As far as most compilers are concerned, it is a valid C (or C++) program. However, it has a number of subtle errors and suspicious constructs that will be reported upon inspection by PC-lint Plus. Here is the default output of PC-lint Plus when processing this example (referred to as `intro_example1.c` below):

```

--- Module:  intro_example1.c (C)
intro_example1.c 6 info 784: nul character truncated from string
    char name[11] = "Joe Jakeson";
                    ^~~~~~

intro_example1.c 22 warning 604: returning address of auto variable 'name'
    return name;
    ^~~~

intro_example1.c 10 warning 530: 'k' is likely uninitialized
    k++;
    ^

intro_example1.c 5 supplemental 891: allocated here
```



```

    int i, k, kk;
    ^
intro_example1.c 8  info 779: string constant in comparison operator '=='
    temp = p == "" ? "null" : p;
    ^  ~

```

The default message format contains the file name and line number corresponding to the diagnostic, the message category (error, warning, etc.), the message number, and the text of the message. Following the main diagnostic line is the line of source code being referenced, followed by the message pointer (indicated by a caret) along with any applicable highlighting (indicated by tildes). We have added blank lines between the messages in this example. In some cases, multiple locations may be relevant to the diagnostic. In this case, the primary location is provided in the main message and one or more supplemental messages (the 891 messages above) follow with the other pertinent locations. In such cases, the supplemental messages should be considered as being "attached" to the immediately preceding primary message.

The format of the messages is fully customizable to allow integration with IDEs that expect a particular diagnostic format or other post-processing tools that require data formatted in XML, JSON, etc. The display of supplemental messages can be disabled, as can the display of extracted source lines and message pointers.

Diagnostics produced by PC-lint Plus fall into one of four categories: error, warning, info, and note. The default warning level is 3, which means that only errors, warnings, and infos are presented. Notes (also referred to as "elective notes" due to the fact that they are disabled by default), can be enabled by setting the warning level to 4 with the option `-w4` or by enabling specific elective notes. Similarly, reducing the warning level or disabling individual messages that are enabled by default can be used to limit message output. Here is an example of some of the elective note messages that are emitted when processing the above example with the highest warning level:

```

intro_example1.c 5  note 9146: multiple declarators in a declaration
    int i, k, kk;
    ^

intro_example1.c 1  note 9403: function 'report' parameter 2 has same
    unqualified type ('short') as previous parameter
char *report(short m, short n, char *p) {
    ~~~~~~ ^

intro_example1.c 8  note 9050: dependence placed on operator precedence
    (operand of ?:)
    temp = p == "" ? "null" : p;
    ^

```

Elective notes are not enabled by default because they do not necessarily represent a fault or likely defect but rather provide certain information that some users find informative.

PC-lint Plus communicates with the programmer via diagnostic messages and the programmer communicates with PC-lint Plus through the use of lint options. Options start with a `+` or `-` (except for the special `!e` option) and can appear on the command line, within lint option files, or in special comments within your source code. Options are processed in the order they are encountered, which means they can be used to temporarily alter the behavior of PC-lint Plus. Options are used to specify all configurable properties of PC-lint Plus, from the location of include directories and pre-defined macros, to which messages should be suppressed and how diagnostics should be communicated.

## 2 Installation and Configuration

### 2.1 System Requirements

#### 2.1.1 Supported Operating Systems

- Windows 7 SP1, Windows 8.1, Windows 10, Windows 11.
- macOS 10.15 and later.
- Linux with kernel 2.6.32 or higher and glibc 2.17 or higher.

#### 2.1.2 Supported File Encodings

Source code files and configuration files processed by PC-lint Plus must employ one of the following encodings:

- UTF-8
- UTF-16
- 7-bit (non-extended) UTF-8 compatible ASCII text

It is not necessary for all files to use the same encoding, e.g. some files may be encoded as UTF-8 while others are encoded as UTF-16. Files not encoded using one of the above encodings should be converted to UTF-8 before being supplied to PC-lint Plus. There are many tools that can perform such conversion including `iconv` for Linux and macOS and `win-iconv` for Windows.

#### 2.1.3 Hardware Requirements

##### Minimum Requirements

- 2 GB RAM
- x86-64 compatible CPU

##### Recommended Requirements

- 2 GB RAM plus 1 GB for every concurrent thread
- Multicore x86-64 compatible CPU

#### 2.1.4 Antivirus Software

Some antivirus software packages employ intrusive mechanisms, such as remote code injection, during their operation. The use of such intrusive mechanisms has the potential to impact the behavior of PC-lint Plus leading to unexpected results. If you experience issues such sluggish performance, application crashes, or inconsistent behavior while using PC-lint Plus in conjunction with such software, you can try running PC-lint Plus with the antivirus software temporarily disabled, with an exception for the PC-lint Plus executable registered with the antivirus software, or on a machine where the antivirus software is not running to help determine if the antivirus software is responsible for the observed behavior.

### 2.2 Installation

PC-lint Plus is distributed as a binary executable and may be installed by copying this file to the desired directory. PC-lint Plus does not require any other installation steps in order to use the product.

If you wish to run PC-lint Plus without specifying the full path of the binary, you will need to either place the PC-lint Plus executable in one of the directories included in your PATH environment variable or add the installed directory to your path.

### 2.2.1 Setting the PATH environment variable on Windows

1. Open a command or run prompt such as by pressing 'R' while holding the Windows key.
2. Type `sysdm.cpl` into the run prompt and press Enter.
3. Select the "Advanced" tab in the dialog window.
4. Click "Environment Variables".
5. In the "System Variables" section, locate the PATH (or Path) variable and select "Edit...".
  - (a) Windows 10: select "New" and type (or "Browse..." for) the directory containing the PC-lint Plus executable.
  - (b) Previous versions of Windows: add the directory containing the PC-lint Plus executable to the semicolon-delimited list.
6. Click "OK" on this and the remaining dialog windows.

Verify the change by running `echo %PATH%` from a newly opened command prompt window.

Note: You may need to have administrative privileges to alter system settings.

Note: This change will not affect already open windows.

### 2.2.2 Setting the PATH environment variable on Linux

1. Open `~/.profile` with your preferred editor
2. Add the line `PATH=$PATH:pc-lint-plus-dir` where `pc-lint-plus-dir` is the full path of the directory where the PC-lint Plus executable is located.
3. Save and quit.

Verify the change by running `echo $PATH` from a newly opened terminal window.

Note: This change will not affect already open terminal windows.

### 2.2.3 Setting the PATH environment variable on macOS

1. Open a terminal such as by pressing the space bar while holding the command key (to open Spotlight) and entering "Terminal"
2. Run the command `sudo nano /etc/paths`
3. At the bottom of the file, add the directory containing the PC-lint Plus binary
4. Press Ctrl-X to quit and press 'Y' to save when prompted.

Verify the change by running `echo $PATH` from a newly opened terminal window.

Note: This change will not affect already open terminal windows.

## 2.3 Configuring with `pclp_config`

### 2.3.1 Overview

PC-lint Plus ships with an automated configuration tool named `pclp_config.py`, which can be found in the `config/` directory of the PC-lint Plus distribution. This Python script simplifies the process of configuring PC-lint Plus for your compiler and project. The next section walks through an example demonstrating why the configuration process is necessary and how to use `pclp_config.py` to generate the appropriate configuration files.

Note: `pclp_config` requires that the Python interpreter as well as the `regex` and `pyyaml` modules to be installed. See [2.3.25 Installing python](#) and [2.3.26 Installing required modules](#) for instructions on installing these components.

### 2.3.2 Introduction and Walkthrough of Automated Configuration with `pclp_config`

In this section we will work with a simple project and generate the configurations necessary for PC-lint Plus to process it. The general process is the same for projects of all shapes and sizes.

Our sample project consists of a single source file, `hello.c`, that contains:

```
#include <stdio.h>

int main(void) {
    printf("Hello, %s\n", SUBJECT);
    return 0;
}
```

and a Makefile that describes how the project should be built:

```
CC = gcc
CFLAGS = -DSUBJECT="\World\"

hello.o: hello.c
    $(CC) $(CFLAGS) -c hello.c -o hello.o
hello: hello.o

clean:
    rm hello.o hello
```

The project is built by running the `make` utility to compile the program based on the instructions provided in the make file. In our case we would build by running `make hello`; running `make clean` will remove object files and executables.

Attempting to lint `hello.c` without any configuration at all (e.g. `pclp hello.c`) produces the following error:

```
hello.c 1 error 322: unable to open include file 'stdio.h'
#include <stdio.h>
      ^
```

The issue is that while your compiler knows where to find your system header files, such as `"stdio.h"`, PC-lint Plus does not. This information, as well as other details about your compiler such as fundamental type sizes and predefined macros, needs to be configured before PC-lint Plus can properly analyze your source code. The process of compiling the necessary information for a particular compiler is referred to as a *compiler configuration*.

Section [2.4 Configuring manually](#) discusses the process of manually extracting the various pieces of information to create a compiler configuration from scratch but in most cases the process can be automated by using `pclp_config`, which is what we will do here.

For this example, our compiler is `gcc`. The first thing we need is the full path of the `gcc` executable. If we do not know this, we can run the command `whereis gcc` or `which gcc` from a command prompt on Linux or macOS and `where gcc` from a command prompt on Windows.

Once we have the location of the compiler, we are ready to generate a compiler configuration using `pclp_config.py`, using the following command:

```
pclp_config.py
--compiler=gcc
--compiler-bin=/usr/bin/gcc
--config-output-lnt-file=co-gcc.lnt
```

```
--config-output-header-file=co-gcc.h
--generate-compiler-config
```

`pclp_config` options start with `--` and arguments are provided to options by separating them with an equal sign (`=`). The first argument tells `pclp_config` the compiler family we are targeting (`gcc`); the `--list-compilers` option can be used to show all supported compiler families. The second option specifies the full path of the compiler (`/usr/bin/gcc`), which `pclp_config` will use to extract the necessary information and build a compiler configuration. The next two options instruct `pclp_config` where to store the configuration (`co-gcc.lnt`) and header file (`co-gcc.h`) it will generate. The names are up to you but the standard convention is `co-COMPILER.lnt` and `co-COMPILER.h`. The last option tells `pclp_config` to generate a *compiler* configuration (as opposed to a *project* configuration, discussed later).

Note: In this walkthrough we assume that the `compilers.yaml` file, the compiler database used by `pclp_config` located in the `config/` directory of the PC-lint distribution, resides in the same directory that `pclp_config` is invoked from. If this is not the case, you will need to add the option `--compiler-database=/path/to/compilers.yaml` to each `pclp_config` command.

Note: If you receive an error about 'utf8' such as "UnicodeDecodeError: 'utf8' codec can't decode" when running `pclp_config` from the Windows command line, you may need to change the code page of that session to UTF-8 using the command `chcp 65001` before running `pclp_config`.

Note: If your compiler or development process contains a "developer prompt" or a script that needs to be run to load appropriate compiler paths or other settings when working from the command line, the commands discussed in this section should be executed from such a prompt after any such start-up scripts have run.

Running this command results in a customized `co-gcc.lnt` file that will look something like this:

```
/* Compiler configuration for gcc 4.8.4.
   Using the options:
   Generated on 2017-05-15 12:53:06 with pclp_config version 1.0.0.
*/

...

// Size Options
-si4 -sl8 -sll8 -ss2 -sw4 -sp8 -sf4 -sd8 -sld16
// Include Options
-i"/usr/lib/gcc/x86_64-linux-gnu/4.8/include"
-i"/usr/local/include"
-i"/usr/lib/gcc/x86_64-linux-gnu/4.8/include-fixed"
-i"/usr/include/x86_64-linux-gnu"
-i"/usr/include"
...
+libh(co-gcc.h)
-header(co-gcc.h)
```

and a `co-gcc.h` file that contains macro definitions extracted from the compiler.

Now that we have a compiler configuration, we can add it to our lint command, before our source file (note that we do not reference `co-gcc.h` directly as a reference to this file, it is automatically included in the generated `co-gcc.lnt` file):

```
pclp co-gcc.lnt hello.c
```

While PC-lint Plus now finds the "stdio.h" header, we encounter a different error:

```
hello.c 4 error 40: undeclared identifier 'SUBJECT'
```

```
printf("Hello, %s\n", SUBJECT);
      ^
```

`SUBJECT` is a macro that is used, but not defined, in the source code. Since the macro is not a predefined compiler macro our compiler configuration does not have the definition either. Instead, `SUBJECT` is a project macro and demonstrates the need for a *project* configuration file.

When your build process builds your project, it often adds compiler options that introduce project-specific macros and include directories as well as other options that affect the behavior of the compiler. In our case, the build system is `make` and the Makefile contains a compiler option that defines this macro (`CFLAGS=-DSUBJECT="\`World`\`")` when the project is built. PC-lint Plus needs this information as well.

Given a set of compiler invocations used to build a project, `pclp_config` can generate a project configuration by interpreting the compiler options present within the invocations and generating a corresponding project configuration file.

**Note:** Many build systems (including `cmake`, `Ninja`, `Meson`, and `Qbs`) support the generation of a JSON compilation database (`compiler_commands.json`). When using such a build system, the recommended way to generate a project configuration is by using this compilation database with `pclp_config` as described [here](#). The remainder of this section describes the process of using the `imposter` program which is provided to assist in the generation of a project configuration when a compilation database is not available.

To obtain the compiler invocations, we need to employ a separate tool, the `imposter`. The `imposter` program, provided as `imposter.c` in the `config/` directory of the PC-lint Plus distribution, is a stand-in for the compiler that logs the options it is called with to a file in a format that can be used by `pclp_config` to generate a project configuration.

After compiling `imposter.c` to `imposter` with a C compiler, you will need to modify your build process by telling it to execute the `imposter` instead of the compiler. Each build system has a different way of doing this, which often involves setting an environment variable or using a command-line option. See the documentation for your build system for details or the examples that appear later for details about how to do this on some of the more common build systems.

In the case of `make`, the location of the C compiler is typically stored in a variable called `CC` (`CXX` is used to store the location of the C++ compiler to use). The `CC` variable is defined at the top of our Makefile. We can either edit this line to point it to our `imposter` binary or override the `CC` variable on the command line using `make -e CC=/path/to/imposter hello`.

Before we run our modified build though, we need to tell the `imposter` program where to write its output (the default is `stdout`). Because `imposter` is a stand-in for the compiler, it needs to handle all the options that might be provided to it by the build process. For this reason, the `imposter` uses environment variables instead of options to affect its behavior. The `IMPOSTER_LOG` environment variable is used to tell `imposter` where to write invocation data. We could modify the Makefile to set this environment variable before invoking the `imposter` but we will set it on the command line so we do not have to modify the Makefile. On Linux or macOS this can be done with:

```
export IMPOSTER_LOG=hello.commands
```

and on Windows:

```
set IMPOSTER_LOG=hello.commands
```

Now we can run `make`:

```
make -e CC=/path/to/imposter hello
```

The file, `hello.commands`, should now contain the compiler invocations in YAML format, looking something like this:

```
['-DSUBJECT="World"', '-c', 'hello.c', '-o', 'hello.o']
['hello.o', '-o', 'hello']
```

We can now use `pclp_config` to convert this file into a project-specific configuration for PC-lint Plus:

```
pclp_config.py
--compiler=gcc
--imposter-file=hello.commands
--config-output-lnt-file=hello.lnt
--generate-project-config
```

As before, we need to tell `pclp_config` what compiler we are using so that it knows how to interpret the options that were logged by the imposter program, which we do with the first option. The second option tells `pclp_config` where to find the logged invocations. The third option specifies where the project configuration should be written. Like the compiler configuration, this name is up to you but is typically named after the project. Finally, the last option specifies the type of configuration to generate. Running this command will create the necessary project configuration in `hello.lnt`, which will look like:

```
-env_push
-dSUBJECT="World"
"hello.c"
-env_pop
```

This file contains everything that is needed to analyze the project, in this case the macro definition that was missing before and the name of the source files that constitute the project (just one in this case). The `-env_push` and `-env_pop` options that surround each source file prevent options, such as `-d`, from extending past the source files they should be applied to (see [-env\\_push](#) and [env\\_pop](#) for more information about these options).

We can now analyze the full project using our generated configuration files:

```
pclp co-gcc.lnt hello.lnt
```

Note that the compiler configuration comes before the project configuration. Also note that we do not need to specify the project's source file on the command line as any source files will be included in the project configuration.

Both the `imposter` program and the `pclp_config` script have a number of configurable options to handle various situations. For example, some projects might not get through the build process without source files being compiled in which case `imposter` needs to be configured to call the actual compiler after it logs the options it was called with. If you are targeting an architecture that is not your compiler's default, you will need to provide the architecture options to `pclp_config` when building the compiler configuration. These details are documented in the [2.3.23 Imposter Options Reference](#) and [2.3.22 pclp\\_config Options Reference](#) sections.

### 2.3.3 Creating a compiler configuration for GCC or Clang

1. Locate the compiler binary. On Linux or macOS this can be done with the command:

```
which gcc
```

and on Windows with the command:

```
where gcc
```

replacing `gcc` with the name of your compiler.

For STM32CubeIDE users, the location of the compiler can be obtained by following the below steps in the IDE:

- (a) Click on "Window" and then "Preferences".
- (b) In the left-hand pane, expand "C/C++", then expand "Build", and then click on "Build Variables".
- (c) The location of your compiler should be represented by either the `gnu_tools_for_stm32_compiler_path` or the `gnu_arm_embedded_compiler_path` variables. You should be able to view the Value of the appropriate variable in the list. If it shows up as "ECLIPSE DYNAMIC VARIABLE" you should be able to see the value being used by selecting the variable and clicking on "Edit".
- (d) Once you have located the directory, you can open it to find your gcc executable, e.g. `arm-none-eabi-gcc.exe`. You can then use the full path (including the directory and the executable file name) as the argument to `--compiler-bin` in the invocation of `pclp_config.py` provided below.

2. Run the `pclp_config.py` script. For GCC use:

```
pclp_config.py
--compiler=gcc
--compiler-bin=/path/to/compiler
--config-output-lnt-file=co-gcc.lnt
--config-output-header-file=co-gcc.h
--generate-compiler-config
```

For clang use:

```
pclp_config.py
--compiler=clang
--compiler-bin=/path/to/compiler
--config-output-lnt-file=co-clang.lnt
--config-output-header-file=co-clang.h
--generate-compiler-config
```

If the `compilers.yaml` file (found in the `config/` directory of the distribution) is not in the directory from which `pclp_config.py` is run, you will need to add the option:

```
--compiler-database=/path/to/compilers.yaml
```

If you are creating a configuration for a hardware target that is not the compiler's default, you will need to add the option:

```
--compiler-options="arch-option"
```

where *arch-option* specifies the target architecture, e.g.:

```
--compiler-options="-m32"
```

If you use a C or C++ standard language version that is not your compiler's default, you will also want to add the appropriate `-std` options using `--compiler-c-options` and `--compiler-cpp-options`, e.g.:

```
--compiler-c-options="-std=c99"
--compiler-cpp-options="-std=c++14"
```

3. Review the generated `.lnt` and `.h` file for completeness.

The generated configuration will be suitable for both C and C++ source files, there is no need to separately configure using `g++` or `clang++`.



### 2.3.4 Creating a compiler configuration for Microsoft C/C++ compilers

1. Open a "Developer Command Prompt for Visual Studio".

In Windows 10:

- Press the Windows key to open the Start menu and type "dev".
- Select the appropriate "Developer Command Prompt" to launch.

In Windows 8.1:

- Press the Windows key to open the Start screen.
- Press CTRL + TAB to open the Apps List.
- Press V to show available Visual Studio command prompt options.
- Select the appropriate "Developer Command Prompt" to launch.

In Windows 8:

- Press the Windows key to open the Start screen.
- Press Z while holding the Windows key.
- Select the "Apps view" icon at the bottom and press V to show the available Visual Studio command prompt options.
- Select the appropriate "Developer Command Prompt" to launch.

In Windows 7:

- Press the Windows key to open the Start menu, select "All Programs", and then expand "Microsoft Visual Studio".
- Select "Visual Studio Command Prompt" from the menu or from the "Visual Studio Tools" menu to launch.

2. Find the location of the `cl.exe` compiler:

From the Developer Command Prompt opened in step 1, run:

```
where cl
```

3. Run the `pclp_config.py` script:

```
python pclp_config.py
--compiler=vs2015
--compiler-bin=/path/to/compiler
--config-output-lnt-file=co-vs2015.lnt
--config-output-header-file=co-vs2015.h
--generate-compiler-config
```

replacing `vs2015`, in all three locations, as appropriate for the version of Visual Studio you are targeting and `/path/to/compiler` with the full path of the `cl.exe` binary located in step 2.

Note that `vs2015`, `vs2017`, etc. are for 32-bit targets, use `vs2015_64`, `vs2017_64`, etc. for 64-bit configurations. The `--list` option can be used to show all supported values for `--compiler`.

If the `compilers.yaml` file (found in the `config/` directory of the distribution) is not in the directory from which `pclp_config.py` is run, you will need to add the option:

```
--compiler-database=/path/to/compilers.yaml
```

See [2.3.25 Installing Python](#) for steps to install Python if necessary.

If you are creating a configuration for a project that has changed the “C++ Language Standard” setting, you will need to add an option of the form:

```
--compiler-options="/std:language-standard"
```

where *language-standard* specifies the language standard argument to the compiler option corresponding to the language standard selection, e.g.:

```
--compiler-options="/std:c++17"
```

specifies C++17. If the command to generate a configuration for a project that uses a non-default language standard does not specify the compiler option corresponding to the language standard then the resulting incomplete configuration may lead to error or message directives and missing standard library types. <sup>1</sup>

4. Review the generated `.lnt` and `.h` file for completeness.

The generated configuration will be suitable for both C and C++ source files.

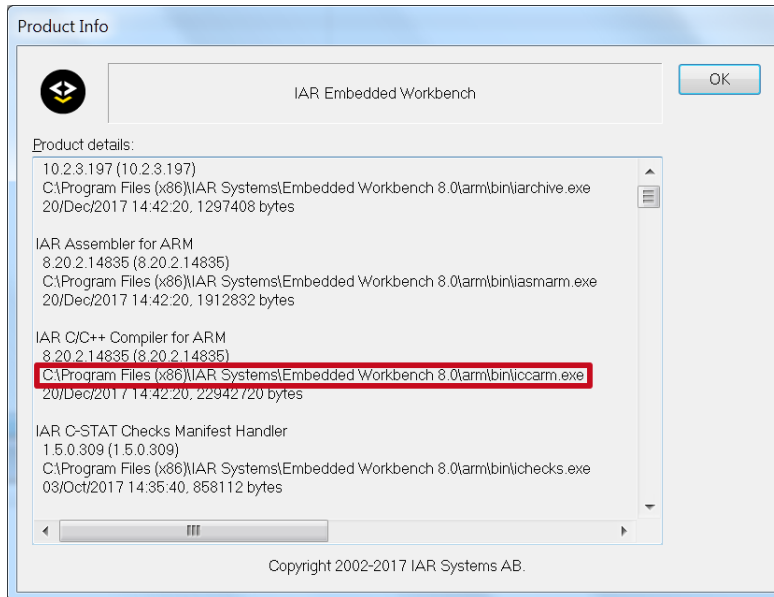
### 2.3.5 Creating a compiler configuration for IAR Embedded compilers

PC-lint Plus supports automated configuration for all IAR compiler families. To generate a compiler configuration for an IAR compiler using `pclp_config`, you will need to determine the value to use for the `--compiler` option. The table below lists the different compiler families and some of the corresponding chip-sets.

Value for <code>--compiler</code> Option	Description
<code>iar-430</code>	IAR compiler for Texas Instruments MSP430 and MSP430X
<code>iar-78k</code>	IAR compiler for Renesas 78K0/78K0S and 78K0R
<code>iar-8051</code>	IAR compiler for the 8051 microcontroller family
<code>iar-arm</code>	IAR compiler for ARM Cores
<code>iar-avr</code>	IAR compiler for Atmel AVR
<code>iar-avr32</code>	IAR compiler for Atmel AVR32
<code>iar-cf</code>	IAR compiler for NXP/Freescale ColdFire
<code>iar-cr16c</code>	IAR compiler for National Semiconductor CR16C
<code>iar-h8</code>	IAR compiler for Renesas H8/300H and H8S
<code>iar-hcs12</code>	IAR compiler for NXP/Freescale HCS12
<code>iar-m16c</code>	IAR compiler for Renesas M16C/1X-3X, 5X-6X and R8C Series
<code>iar-m32c</code>	IAR compiler for M32C and M16C/8x Series
<code>iar-maxq</code>	IAR compiler for Dallas Semiconductor MAXQ
<code>iar-r32c</code>	IAR compiler for Renesas R32C/100 microcomputer
<code>iar-rh850</code>	IAR compiler for Renesas RH850
<code>iar-rl78</code>	IAR compiler for Renesas RL78
<code>iar-rx</code>	IAR compiler for Renesas RX
<code>iar-s08</code>	IAR compiler for NXP/Freescale S08
<code>iar-sam8</code>	IAR compiler for Samsung SAM8
<code>iar-v850</code>	IAR compiler for Renesas V850

<sup>1</sup>For example, if a Visual Studio project that uses the `/std:c++17` compiler option omits the option `--compiler-options="/std:c++17"` when generating the compiler configuration then the type `std::optional` from the `<optional>` header will be disabled by a version check within the standard library and the message `class template optional is only available with C++17 or later.` will be emitted by a preprocessing directive. This also applies to other C++17 library features like `std::any`.

You will also need the full path of the compiler which can be obtained from within the IAR Embedded Workbench by clicking "Help", "About", "Product Info", then clicking on the "Details" button which should bring up a dialog like the one shown below:



For example, the following command will create a configuration for the IAR ARM compiler, located in `C:\iar\arm\iccarm.exe`, with output files named `co-iar-arm.lnt` and `co-iar-arm.h`:

```
python pclp_config.py
    --compiler=iar-arm
    --compiler-bin=C:\iar\arm\iccarm.exe
    --config-output-lnt-file=co-iar-arm.lnt
    --config-output-header-file=co-iar-arm.h
    --compiler-options="..."
--generate-compiler-config
```

The `--compiler-options` option specifies the options passed to the IAR compiler when compiling your project, make sure to replace `...` with the options used to compile your project. If you are generating a configuration for use with C++ source files, make sure to include the option `--c++` with `--compiler-options`, e.g. `--compiler-options="--c++"`.

If the `compilers.yaml` file (found in the `config/` directory of the distribution) is not in the directory from which `pclp_config.py` is run, you will need to add the option:

```
--compiler-database=/path/to/compilers.yaml
```

See [2.3.25 Installing Python](#) for steps to install Python if necessary.

The generated configuration will be suitable for both C and C++ source files.

See [2.3.15 Integrating PC-lint Plus with IAR Embedded Workbench](#) for instructions on integrating PC-lint Plus with the IAR IDE.

### 2.3.6 Creating a compiler configuration for Keil $\mu$ Vision ARMCC

1. From the "Help" menu, select "About  $\mu$ Vision".
2. The dialog will list the "Toolchain Path" and the "Compiler". The full compiler path is the toolchain path followed by the compiler name. For example, `"C:\Keil_v5\ARM\ARMCC\bin\armcc.exe"`.

3. From the "Project" menu, select "Options [for target]".
4. Select the "C/C++" tab.
5. Compiler options are listed in the "Compiler control string" field, for example `--c99 -c --cpu Cortex-M7 --fpu=SoftVFP`.
6. The compiler configuration for this example could be generated by running:

```
python pclp_config.py
--compiler=keil_armcc
--compiler-bin="C:\Keil_v5\ARM\ARMCC\bin\armcc.exe"
--config-output-lnt-file=co-keil.lnt
--config-output-header-file=co-keil.h
--compiler-options="--c99 -c --cpu Cortex-M7 --fpu=SoftVFP"
--generate-compiler-config
```

Note: If the configuration files are not placed in the project working directory the option `--header-option-use-enclosing-directory` can be added when running `pclp_config` so that the header file can be found in the same directory as the options file regardless of where the options file is used.

See [2.3.16 Integrating PC-lint Plus with Keil µVision](#) for instructions on integrating PC-lint Plus with the Keil IDE.

### 2.3.7 Creating a compiler configuration for Microchip MPLAB X XC8/XC16/XC32

1. In the "Run" (or "Production") menu, select "Clean and Build Main Project"
2. In the output pane, look for a line like:  
`"C:\Program Files\Microchip\xc32\v1.40\bin\xc32-gcc.exe" -x c -mprocessor=32MX795F512L`  
 Note that the `-mcpu=` option may appear instead of `-mprocessor=`. In this example, the compiler binary is `C:\Program Files\Microchip\xc32\v1.40\bin\xc32-gcc.exe` and the relevant compiler options are `-x c` (setting the language to C, as opposed to C++) and `-mprocessor=32MX795F512L` setting the target. Relevant compiler options should be passed to `pclp_config.py` using the `--compiler-options` option. The binary name identifies the compiler architecture (32-bit in this example).
3. The configuration for this example would be generated with the command:

```
python pclp_config.py
--compiler=microchip_xc32
--compiler-bin="C:\Program Files\Microchip\xc32\v1.40\bin\xc32-gcc.exe"
--config-output-lnt-file=co-xc.lnt
--config-output-header-file=co-xc.h
--compiler-options="-x c -mprocessor=32MX795F512L"
--generate-compiler-config
to produce a compiler configuration.
```

Note: If the configuration files are not placed in the project working directory the option `--header-option-use-enclosing-directory` can be added when running `pclp_config` so that the header file can be found in the same directory as the options file regardless of where the options file is used.

See [2.3.17 Integrating PC-lint Plus with MPLAB X](#) for instructions on integrating PC-lint Plus with the MPLAB X IDE.

### 2.3.8 Creating a compiler configuration for MetaWare ccac ARC V2 EM/V2 HS/V3 HS/etc.

1. From a Command Prompt or terminal, run `where ccac` (Windows) or `which ccac` (Linux) to obtain the compiler path. If the compiler is not in your system path then you will need to find the full

path of `ccac`. In a typical installation, it will be located in `ARC/MetaWare/arc/bin` relative to the installation root.

2. Open the MetaWare IDE. In the “Project” menu, select “Clean Project”.
3. In the “Project” menu, select “Build Project”.
4. In the “Console” pane, look for lines like:

```
Invoking: MetaWare ARC EM C/C++ Compiler
ccac -c -g -Hnocopyr -Hpurge -arcv2em -core1 -o "example.o" "../example.c"
Relevant compiler options should be passed to pclp_config.py using the --compiler-options option.
```

If you do not see an invocation of the `ccac` compiler, then your project may use `mcc` or `gcc`. For `gcc`, see [2.3.3 Creating a compiler configuration for GCC or Clang](#). Projects that require the unique extensions of the `mcc` compiler are not supported. A compiler database entry for the older `mcc` compiler is not provided as the unique language extensions in the `mcc` compiler that are not supported by the `ccac` compiler are not currently supported by PC-lint Plus. While the manual configuration steps documented in section [2.4 Configuring manually](#) can be applied to the `mcc` compiler, the outcome will depend on the degree of reliance on major extensions. Information about MetaWare extensions supported only in the `mcc` compiler can be found in the “DesignWare MetaWare C/C++ Language Reference” in sections 2.3 (MetaWare Extensions) and 3 (Iterators in MetaWare C/C++).

5. The configuration for this example would be generated with the command:

```
python pclp_config.py
--compiler=metaware_ccac
--compiler-bin=/home/example/ARC/MetaWare/arc/bin/ccac
--config-output-lnt-file=co-mw.lnt
--config-output-header-file=co-mw.h
--compiler-options="-arcv2em -core1"
--generate-compiler-config
to produce a compiler configuration.
```

Note: If the configuration files are not placed in the project working directory the option `--header-option-use-enclosing-directory` can be added when running `pclp_config` so that the header file can be found in the same directory as the options file regardless of where the options file is used.

See [2.3.18 Integrating PC-lint Plus with Eclipse-based IDEs](#) for instructions on integrating PC-lint Plus with MetaWare IDE.

### 2.3.9 Creating a compiler configuration for Texas Instruments Code Composer Studio compilers

1. Open the project in Code Composer Studio. In the “Project” menu, select “Clean...”.
2. Uncheck both “Clean all projects” and “Start a build immediately”.
3. Ensure that the correct project to clean is selected.
4. Press “Clean”.
5. In the “Project” menu, select “Build Project”.
6. In the “Console” pane, look for an invocation of `armcl`, `cl430`, `cl2000`, or `cl6x` similar to:

```
"/ti/ccs0000/ccs/tools/compiler/compiler-family/cl430" --define=family -i"directory"
--include_path="directory"
```

 Relevant compiler options such as macro definitions and include options should be passed to `pclp_config.py` using the `--compiler-options` option.

If you instead find an invocation of `gcc`, see [2.3.3 Creating a compiler configuration for GCC or Clang](#).

7. The configuration for this example would be generated with the command:

```
python pclp_config.py
  --compiler=ti_cl430
  --compiler-bin=/ti/ccs0000/ccs/tools/compiler/compiler-family/cl430
  --config-output-lnt-file=co-msp430.lnt
  --config-output-header-file=co-msp430.h
  --compiler-options='--define=family -i"directory" --include_path="directory"'
  --generate-compiler-config
to produce a compiler configuration.
```

- Note: The configuration command includes a single-quoted string with inner double quotes. The Windows Command Prompt does not recognize single quotes, requiring that they are replaced with double quotes and each inner double quote escaped with a backslash. It is recommended on Windows to instead execute the command using the single quote support available in PowerShell.

Texas Instruments compilers available for use with `pclp_config.py`:

Value for <code>--compiler</code>	Description
<code>ti_cl430</code>	Texas Instruments <code>cl430</code> for MSP430
<code>ti_cl2000</code>	Texas Instruments <code>cl2000</code> for C2000
<code>ti_cl6x</code>	Texas Instruments <code>cl6x</code> for C6000
<code>ti_armcl</code>	Texas Instruments <code>armcl</code> for ARM

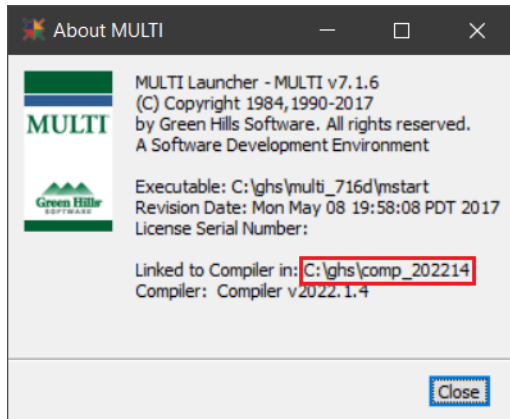
Note: `cl6x` vector extensions are not supported.

Note: If the configuration files are not placed in the project working directory the option `--header-option-use-enclosing-directory` can be added when running `pclp_config` so that the header file can be found in the same directory as the options file regardless of where the options file is used.

See [2.3.18 Integrating PC-lint Plus with Eclipse-based IDEs](#) for instructions on integrating PC-lint Plus with Code Composer Studio.

### 2.3.10 Creating a compiler configuration for Green Hills Software compilers

PC-lint Plus supports automated configuration for all Green Hills Software (GHS) compiler families. To generate a compiler configuration for a GHS compiler using `pclp_config`, you will need to determine the full path of the compiler driver. One method to determine the path begins by getting the compiler driver's directory using the GHS MULTI Launcher. Within the GHS MULTI Launcher, select "Help" and then "About MULTI Launcher..." which should bring up a dialog like the one shown below:



The value of the "Linked to Compiler in:" field indicates the directory where the compiler driver is located. You will need to examine this directory to determine the name of the compiler driver. The compiler driver typically is named `cc` followed by the processor family name. For instance, the ARM compiler is named `ccarm` (32-bit) or `ccarm64` (64-bit). You can find the compiler name for a specific source file by selecting it in the MULTI Project Manager and consulting the Command pane.

The complete path of the compiler driver is provided to `pclp_config` as the value of the `--compiler-bin` option. For example, the following command will create a configuration for the GHS ARM 64-bit compiler driver with a full path of "`c:\ghs\comp_202214\ccarm64.exe`", with output files named `co-ghs-arm64.lnt` and `co-ghs-arm64.h`:

```
python pclp_config.py
    --compiler=ghs
    --compiler-bin=C:\ghs\comp_202214\ccarm64.exe
    --config-output-lnt-file=co-ghs-arm64.lnt
    --config-output-header-file=co-ghs-arm64.h
    --generate-compiler-config
```

If the `compilers.yaml` file (found in the `config` directory of the distribution) is not in the directory from which `pclp_config.py` is run, you will need to add the option:

```
--compiler-database=/path/to/compilers.yaml
```

The generated configuration will be suitable for both C and C++ source files.

### 2.3.11 Creating a compiler configuration for SEGGER compilers

**NOTE:** This tutorial is intended for projects that make use of `gcc` or `SEGGER` as their compiler. Other compilers that may be used for a SEGGER project are not currently supported.

1. Open the desired Solution by opening the "File" menu, then selecting "Open Solution".
2. If a Project Explorer panel is not already open, select the "View" menu.
3. Under the "Project" header, select "Project Explorer".
4. Right click on the desired Project and select "Export Build".
5. This will open a `.txt` file in the editor containing the filepath of the compiler executable.
6. Identify the line that is responsible for compiling the "main" file in your project. It should begin with a file path like: "`C:\Program Files\SEGGER\SEGGER Embedded Studio for ARM 7.32\gcc\arm-none-eabi\bin\cc1`". The text after the final separator in the path should be one of: `cc1`, `cc1plus`, or `segger-cc`. These are also the names that should be passed to the `--compiler=` option.

Compiler options that affect the parsing step of the compiler, such as macro definitions and include options, should be passed to `pclp_config.py` by using the `--compiler-options` option.

7. If creating a configuration for `segger-cc`:

- (a) The `-cc1` option must be included and should be first in the list of options passed to `--compiler-options`.
- (b) The `-triple` option must also be included in the list passed to `--compiler-options`.

8. The compiler configuration for this example can be generated by running the following command:

```
python pclp_config.py
--compiler=cc1
--compiler-bin="C:\Program Files\SEGGER\SEGGER Embedded
  Studio for ARM 7.32\gcc\arm-none-eabi\bin\cc1"
--compiler-options="..."
--config-output-lnt-file=co-segger-cc1.lnt
--config-output-header-file=co-segger-cc1.h
--generate-compiler-config
```

See [2.3.20 Integrating PC-lint Plus with SEGGER IDE](#) for instructions on integrating PC-lint Plus with the SEGGER IDE.

### 2.3.12 Creating a project configuration with a JSON compilation database

The simplest and easiest way to generate a project configuration is by using the JSON compilation database created by your build system (typically named `compile_commands.json`). The same `pclp_config` utility used to generate your compiler configuration can generate a complete project configuration from this file using the following command:

```
python pclp_config.py
--compiler=gcc
--compilation-db=compile_commands.json
--config-output-lnt-file=project.lnt
--generate-project-config
```

replacing `gcc` with the compiler-family of your compiler and `compile_commands.json` with the name of your compilation database. The project may then be analyzed by running PC-lint Plus with the arguments `compiler.lnt project.lnt` where `compiler.lnt` is your compiler configuration.

If your existing build process does not already produce a compilation database, you can instruct it to do so using the instructions below depending on your build system. If your build system is not capable of producing a compilation database, you can either use a third-party tool such as [Bear](#) to generate one or use the provided `imposter` program as described in the following two sections to achieve a similar result.

#### Generating a compilation database with `cmake`

When using Ninja or a Makefile generator, `cmake` can produce a compilation database by using the option `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` when building.

#### Generating a compilation database with `iarbuild`

Recent versions of `iarbuild` can generate a compilation database using the `-jsondb` option, e.g.:

```
iarbuild project.ewp -jsondb config -output compile_commands.json
```



Where *project.eup* is the IAR Embedded Workbench project file and *config* is the name of the build configuration (Release, Debug, etc).

### Generating a compilation database with Meson

Meson automatically generates `compile_commands.json`.

### Generating a compilation database with Ninja

Ninja can generate a compilation database using the command:

```
ninja -t compdb > compile_commands.json
```

since version 1.10. For versions of Ninja between 1.2 and 1.9, a rule name is required, e.g.:

```
ninja -t compdb rule1 [rule2...] > compile_commands.json
```

### Generating a compilation database with Qbs

The Qbs build system can generate a compilation database with the command:

```
qbs generate --generator clangdb
```

### Generating a compilation database with SCons

Since version 4.0, SCons can generate a compilation database using the instructions provided [here](#).

#### 2.3.13 Creating a project configuration with make or cmake

1. Open a command prompt and locate the compiler used in your build process with the `which` or `where` command as shown in step #1 of [2.3.3 Creating a compiler configuration for GCC or Clang](#).
2. Set the `IMPOSTER_LOG` environment variable to the full path of the file that will hold compiler invocations. On most shells this command will look like:

```
export IMPOSTER_LOG=/path/to/imposter-log
```

The file name is not important. If the file already exists, it should be truncated before continuing as `imposter.exe` appends entries to this file without truncating it.

3. From a clean build directory, run your build process using the compiled `imposter` program as the compiler. This is generally accomplished by overriding the `CC` or `CXX` make variables. For `make`, this can be done with the command:

```
make -e CC=/path/to/imposter ...
```

You may also be able to set the `CC` or `CXX` environment variables on the command line before running the build process:

```
export CC=/path/to/imposter
```

You may need to build the project from scratch in a new directory for `make/cmake` to use the new setting.

Note: If your project fails to properly build using `imposter` as the compiler you may need to have `imposter` run the compiler during the build process. This can be accomplished by running the following command and then running the build process:

```
export IMPOSTER_COMPILER=/path/to/compiler
```

Note: If your project contains C and C++ modules and uses a different compiler for each, using `IMPOSTER_COMPILER` will not be sufficient since this solution supports only one compiler. In this case, you will need to set `IMPOSTER_COMPILER_ARG1` (to any value), such as with the command:

```
export IMPOSTER_COMPILER_ARG1
```

And set the `CC` and `CXX` variables like so:

```
export CC="/path/to/imposter /path/to/c-compiler"
export CXX="/path/to/imposter /path/to/c++-compiler"
```

Setting `IMPOSTER_COMPILER_ARG1` will cause `imposter` to use the first argument it is called with as the compiler to invoke, which allows multiple compilers to be supported in a single build. Note that `IMPOSTER_COMPILER` must not be set for this to work.

Note: When using CMake it is often necessary to use `imposter` as the compiler during the project generation step. If the CMake configuration process causes test files to be compiled to assess compiler features you will need to clear `IMPOSTER_LOG` before running the build process to avoid including them in your project configuration.

4. Run `pclp_config` to generate a project configuration using the compiler invocations logged by `imposter`:

```
python pclp_config.py
--compiler=gcc
--imposter-file=/path/to/imposter-log
--config-output-lnt-file=project.lnt
--generate-project-config
```

Replacing `gcc` with the appropriate compiler name, `/path/to/imposter-log` with the same value that `IMPOSTER_LOG` was set to above and `project.lnt` with the desired value.

PC-lint Plus can now analyze your project by running `lint compiler.lnt project.lnt`.

#### 2.3.14 Creating a project configuration with MSBuild on Windows

Note: It is assumed that you have compiled the `imposter.c` file provided in the `config/` directory of the PC-lint Plus distribution to an executable called `imposter.exe`.

1. Follow steps #1 and #2 from Creating a compiler configuration for Microsoft C/cpp compilers to open a Developer Command Prompt and locate the `cl.exe` binary.
2. Locate the Visual Studio project or solution file for your project. Solution files have a `.sln` extension.
3. In the Developer Command Prompt, set the `IMPOSTER_LOG` environment variable to the full path where compiler invocations should be logged:

```
set IMPOSTER_LOG=/path/to/imposter-log
```

The file name is not important. If the file already exists, it should be truncated before continuing as `imposter.exe` appends entries to this file without truncating it.

4. Run `msbuild` on your project file using `imposter.exe` as the compiler by executing the following commands in the same Developer Command Prompt:

```
msbuild project.sln /t:clean
msbuild project.sln /p:CLToolExe=imposter.exe /p:CLToolPath=C:\path\to\imposter
```

Note that the name without a path is provided to the `/p:CLToolExe` option and the path, without the file name, is provided to the `/p:CLToolPath` option.

Note: If your project fails to properly build using `imposter.exe` as the compiler you may need to have `imposter.exe` run the compiler during the build process. This can be accomplished by running the following command and then running the two above commands in the same Developer Command Prompt:

```
set IMPOSTER_COMPILER=/path/to/cl.exe
```

5. Run `pclp_config.py` to process the output of the imposter log and generate a project configuration:

```
python pclp_config.py
--compiler=vs2015
--imposter-file=/path/to/imposter-log
--config-output-lnt-file=project.lnt
--generate-project-config
```

Replacing `vs2015` with the appropriate compiler name, `/path/to/imposter-log` with the same value that `IMPOSTER_LOG` was set to above and `project.lnt` with the desired value.

PC-lint Plus can now analyze your project by running `lint compiler.lnt project.lnt`.

### 2.3.15 Integrating PC-lint Plus with IAR Embedded Workbench

By integrating PC-lint Plus with IAR Workbench, you can run PC-lint Plus from within the IDE and click on the locations given in the message text to navigate to the provided location. To integrate PC-lint Plus with IAR Workbench, you will need the `env-iar.lnt` file (found in the `lnt` directory in the PC-lint Plus distribution); this setup assumes you have copied the file to the same directory as your IAR compiler configuration files. See [2.3.5 Creating a compiler configuration for IAR Embedded compilers](#) for instructions on creating a compiler configuration for your IAR compiler. The following steps will configure IAR Workbench to run PC-lint Plus:

1. In the Tools menus, select "Configure Tools".
2. In the configuration dialog box that opens up, click "New".
3. For "Menu Text" enter "Analyze Current File with PC-lint Plus", you might want to add text to indicate the chipset if you will be configuring PC-lint Plus for multiple IAR compilers.
4. For "Command", enter the location of the PC-lint Plus binary.
5. For "Argument", enter the following:

```
-u -iconfig-loc env-iar.lnt iar-lint-config $FILE_PATH$
```

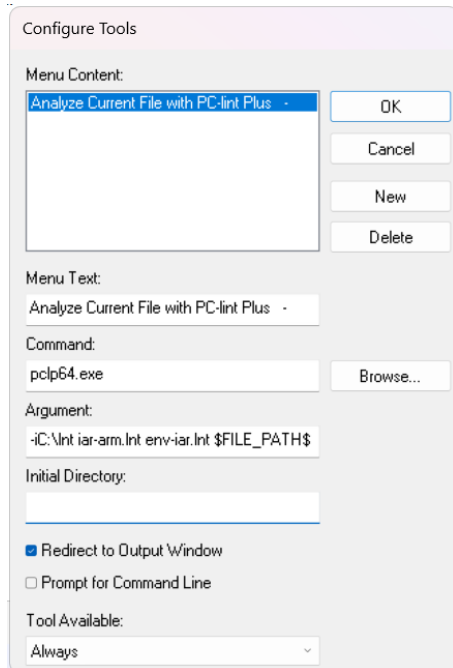
replacing `config-loc` with the full path of the directory containing your configuration files and `iar-lint-config` with the name of your compiler configuration file, e.g. `co-iar-arm.lnt`.

Note: If `config-loc` contains spaces, you will need to include quotes around the argument, e.g.:

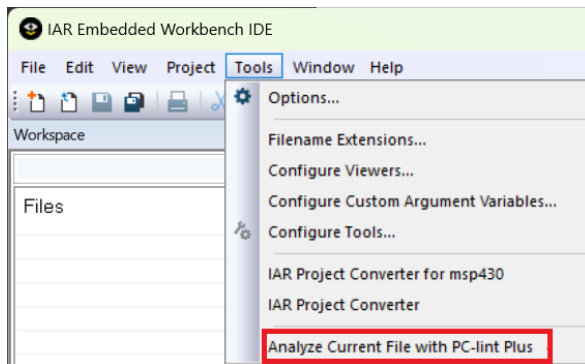
```
"-iC:\location with spaces\lint"
```

6. Check the box "Redirect to Output Window".
7. Select "Always" from the "Tool Available" dropdown.
8. Click on "OK".

The result should look something like:



To run PC-lint Plus from within IAR Workbench on the current file, select the newly added entry from the Tools menu:



To run PC-lint on the full project, follow the above instructions to create a new Tool but for "Menu Text" use "Analyze Project with PC-lint Plus" and for the Argument, replace `$FILE_PATH$` with `$PROJ_DIR$*.c`.

If you have a project-specific configuration file, you can add this at the end of the command line in step #4. If the configuration file contains the list of source files that PC-lint Plus should process, remove the `$FILE_PATH$/$PROJ_DIR$.c` portion from the "Argument" in step #5.

Note: The `env-iar.lnt` file is provided with PC-lint Plus and contains special control characters necessary for proper formatting in IAR Workbench to make locations referenced in messages "clickable". Making modifications to this file may break this formatting.

Assigning a keyboard shortcut to easily run PC-lint Plus:

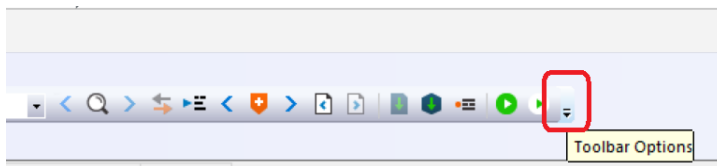
1. In the Tools menus, select "Options".
2. From the left side bar select "Key Bindings".
3. From the "Menu" drop down select "Tools".

4. Scroll down to locate "Analyze Current File with PC-lint Plus" or the name you choose from the previous section.
5. Click on the name and then click on the empty section of "press shortcut key".
6. You can now click the keys to associate with running PC-lint Plus e.g. "Ctrl + Shift + P".
7. Click on "Set" and then "OK".

Now you can run PC-lint Plus on a file by pressing the keyboard shortcut assigned.

Adding PC-lint Plus to the "Toolbar":

1. Click on "Toolbar Options" found here.



2. Hover over "Add or Remove Button" and then click on "Customize".
3. From the "Categories" choose "Tools"
4. On the right-hand side locate "Analyze Current File with PC-lint Plus" or the name you choose from the previous section.
5. Drag and drop the name of the external tool to the "Toolbar".

The result should look something like:



### 2.3.16 Integrating PC-lint Plus with Keil uVision

To set up PC-lint Plus within Keil uVision:

1. In the "Tools" menu, select "Set-up PC-Lint..."
2. An error message will likely appear informing you that uVision could not automatically locate the PC-lint Plus executable. This can be ignored as it will be selected in the next step.
3. In the "Lint Executable" field, browse for your PC-lint Plus executable.
4. Uncheck the "Add Compiler Config" checkbox as the configuration files included with uVision were intended for older PC-lint products.
5. In the "Config File" field, browse for the compiler configuration you generated in the previous section.
6. In the "Include Project Information" group, all of the items that are not disabled can be checked.
7. In both the "C" and "C++" tabs, set "Rules" to "<not used>" and ensure the other fields within the tab are empty.
8. Press "OK" to dismiss the dialog.

To run PC-lint Plus within Keil  $\mu$ Vision:

- In the "Tools" menu, select "Lint"
- In the "Tools" menu, select "Lint All"

### 2.3.17 Integrating PC-lint Plus with MPLAB X

PC-lint Plus can be used from within MPLAB X using a plugin provided by Microchip. To install the plugin:

1. In the "Tools" menu, select "Plugins"
2. From the "Available Plugins" tab, check the "Install" checkbox next to "PCLint"
3. Click the "Install" button

To configure the plugin:

1. In the "Tools" menu, select "Options"
2. Select the "Embedded" Tab in the top row, then the "PCLint" tab
3. In the "Location" field, browse for your PC-lint Plus executable
4. In the "Options" field, select "UserOptionFile"
5. In the "User Option File" field, browse for the compiler configuration you generated in the previous section.
6. In the "Standards" field, select "NOSTANDARD"
7. Press "OK" to dismiss the dialog

To run PC-lint Plus within MPLAB X using the configured plugin:

- Right click a source file in the project list and select "Lint this file"
- Right click a project in the project list and select "Lint this Project"
- Right click within the code editor for a source file and select "Lint this File"

**Note:** This plugin was developed by a third party for an older PC-lint product. Some features of the plugin, such as the coding standard menu, use outdated configuration files. If you intend to use a MISRA standard with MPLAB X, use the latest version of the configuration file included with PC-lint Plus. Coding standard configuration files can be added to the configuration file specified in the "Options" dialog to use them with the plugin.

### 2.3.18 Integrating PC-lint Plus with Eclipse-based IDEs

1. Open the IDE.
2. Select "Run", "External Tools", "External Tools Configurations...".
  - Note: If "External Tools" is not listed in the "Run" menu:
    - (a) Select "Window", "Perspective", "Customize Perspective..."
    - (b) Select the "Menu Visibility" tab.
    - (c) Expand "Run", scroll down, and check the box next to "External Tools". Resume from step 2.
      - Note: If "External Tools" is not listed inside the "Run" tree in the "Menu Visibility" tab, then open the "Window" menu, select "Preferences", expand "General", "Capabilities", and ensure "ANT Tools" is checked. Resume from step (a).
3. Select "Program" in the left sidebar, then press the "New" icon above the list view.

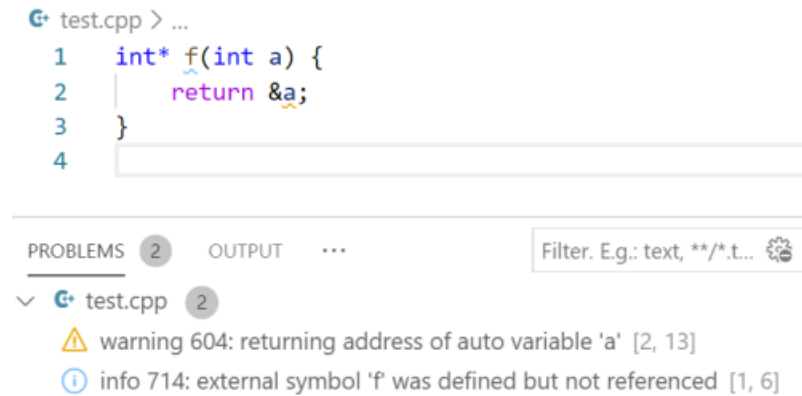
4. In the "Name" field, enter "Run PC-lint Plus on Selected File"
5. For the "Location" field, select "Browse File System..." and locate your PC-lint Plus executable.
6. In the "Working Directory" field, enter `${container_loc}`
7. In the "Arguments" field, add a new line containing `-i` followed immediately (without an intervening space) by the full path to the directory containing your compiler configuration. E.g. `-i/home/example/pclp/config`. Eclipse allows spaces in arguments that are surrounded by double quotes.
8. In the "Arguments" field, add a new line containing the file name of your compiler configuration, e.g. `co-example.lnt`. This file should exist in the directory provided in the previous step.
9. In the "Arguments" field, add a new line and enter `${resource_loc}`
10. Select the "Common" tab and ensure that the "External Tools" checkbox in "Display in favorites menu" is checked.
11. Select "Apply", and then select "Close".
12. Select a source file (not a header file) in the "Project Explorer" pane.
13. Select "Run", "External Tools", "Run PC-lint Plus on Selected File". Output will appear in the "Console" pane.

To analyze a project rather than a single file, follow the above steps but substitute `${resource_loc}/*.c` for `${resource_loc}` when entering the compiler arguments. To run with this external tool configuration, the project directory containing the source files must be selected in the "Project Explorer" pane instead of a source file.

Note: If you receive a variable expansion error from Eclipse when running the external tool in the last step, then ensure that the most recent action within the IDE prior to opening the "Run" menu was to select and focus the target in the "Project Explorer" pane.

### 2.3.19 Integrating PC-lint Plus with Visual Studio Code

The built-in Visual Studio Code “Tasks” feature can be used to integrate PC-lint Plus without using an extension. This can be used to provide the full output in the “Terminal” window and the issued messages in the “Problems” window with code underlining, e.g.:



The Visual Studio Code `tasks.json` file, `settings.json` file, and a PC-lint Plus configuration that can be used to integrate PC-lint Plus as a Task are provided below.

Note that you will still need a compiler and project configuration as described in the enclosing subsection.

`vscode.lnt`:

```
// See https://go.microsoft.com/fwlink/?LinkId=733558
// for documentation on the tasks.json format

// Options for Visual Studio Code compatibility
-width=0                // no line wrapping
-h1                    // set the message height to 1
-format=%f:%l:%C: %t: %t %n: %m" // match compiler message format
+ffn                    // always emit absolute paths
```

`settings.json`:

```
{
  "pclp_exe": "C:\\path\\to\\pclp64.exe",
  "vscode_lnt": "C:\\path\\to\\vscode.lnt",
  "terminal.integrated.scrollback": 3000,
  "win_command": "| ForEach-Object { '{0}.{1}' -f ++$i, $_ -replace '\\.:', '.$${file}:'}",
  "mac_command": "| nl -w1 -s. | sed 's#\\.:#.$${file}:#g'",
  "linux_command": "| nl -w1 -s. | sed 's#\\.:#.$${file}:#g'"
}
```



tasks.json:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "icon": { "color": "terminal.ansiWhite", "id": "search" },
      "presentation": {
        "reveal": "silent",
        "revealProblems": "always",
        "close": false,
        "clear": true,
        "panel": "dedicated",
        "showReuseMessage": false
      },
      "runOptions": { "reevaluateOnRerun": true },
      "label": "PC-lint Plus",
      "type": "shell",
      "detail": "Analyze current file with PC-lint Plus.",
      "windows": {
        "command": "${config:pclp_exe} ${config:vscode_lnt} '${file}' ${config:win_command}",
        "options": { "shell": { "executable": "powershell" } }
      },
      "osx": {
        "command": "${config:pclp_exe} ${config:vscode_lnt} '${file}' ${config:mac_command}"
      },
      "linux": {
        "command": "${config:pclp_exe} ${config:vscode_lnt} '${file}' ${config:linux_command}"
      },
      "problemMatcher": {
        "owner": "C/C++",
        "fileLocation": [
          "absolute"
        ],
        "severity": "info",
        "source": "ID:",
        "pattern": {
          "regexp": "^((\\d+).(.*):(\\d+):(\\d+):\\s+(warning|error|info|note|supplemental):\\s+(.*)$)",
          "code": 1,
          "file": 2,
          "line": 3,
          "column": 4,
          "severity": 5,
          "message": 6
        }
      }
    }
  ]
}
```

To add the Task on a user level:

1. Open the IDE.
2. From the menu bar, select "File", "Preferences", "User Tasks".
  - **Note:** You might need to select "Others" if a dropdown menu appears.
3. Copy the content of `tasks.json` mentioned previously and add it to the `tasks.json` file that is now open on the IDE.
  - **Note:** The previous content in the `tasks.json` can be removed since it shows an example unless you have defined additional tasks in which case PC-lint Plus can be added to the list.
4. From the menu bar, select "File", "Save" to save the changes.
5. From the menu bar, select "View", "Command Palette..."
6. In the empty field, type in "Open User Settings (JSON)". Then select the option once found.
7. Copy the lines from `settings.json` mentioned previously and add it to the `settings.json` file that is now open in the IDE.
  - **Note:** "C:\\path\\to\\pclp64.exe" and "C:\\path\\to\\lnt\\vscode.lnt" should be replaced with the PC-lint Plus executable location and the location of the `vscode.lnt` mentioned above respectively.
8. From the menu bar, select "File", "Save" to save the changes.

**Note:** This task can be added on a project level by creating a folder named `.vscode` in the project directory. Inside the `.vscode` folder, create two additional files named `settings.json` and `tasks.json` with their content being what is described previously.

To bind a keyboard shortcut to easily run the Task:

1. From the menu bar, select "View", "Command Palette..."
2. In the empty field, type in "Open Keyboard Shortcuts (JSON)". Then select the option once found.
3. Add the following to the file:

```
[{
  "key": "ctrl+shift+alt+p",
  "command": "workbench.action.tasks.runTask",
  "args": "PC-lint Plus"
}]
```

This will allow the Task to run by clicking "ctrl+shift+alt+p".

To test the Task, start by opening a `test.c` file and running the Task by using the keyboard shortcut. Alternatively the Task can run by following these steps:

1. From the menu bar, select "View", "Command Palette..."
2. In the empty field, type in "Tasks: Run task". Then click on the option once found.

3. Select "PC-lint Plus".

To enhance the experience of using this task, the third-party extension [Error Lens](#) can be used to better highlight the location of issued messages and automatically display the text of the message.

### 2.3.20 Integrating PC-lint Plus with SEGGER IDE

**Note:** This tutorial is intended for SEGGER Embedded Studio Version 5.68 or newer.

By integrating PC-lint Plus with SEGGER IDE, you can run PC-lint Plus from within the IDE and click on the locations given in the message text to navigate to the provided location. See [2.3.11 Creating a compiler configuration for SEGGER compilers](#) for instructions on creating a compiler configuration for your SEGGER compiler. The following steps will configure the SEGGER IDE to run PC-lint Plus:

1. Select the "Tools" menu.
2. Within the "Tools" menu, select "Options".
3. Within "Options", select the "Building" tab.
4. Scroll down to find "Global Macros", double click it.
5. In the text box, define a new global macro called `LintDir`. This macro will expand to the absolute path of the directory containing the PC-lint Plus executable. If this path includes spaces, do not surround the path with quotes, this will be taken care of in a later step. The definition of the macro should adhere exactly to the following format: `LintDir=path/to/lint/directory`.

- (a) A definition of `LintDir` on Windows would look like:  
`LintDir=C:\Users\user\segger-projects\lint-dir`
- (b) On Linux, this would look like:  
`LintDir=/home/user/segger-projects/lint-dir`

Additionally, the `.lnt` and `.h` files generated in [2.3.11](#) should be placed in a folder called `lnt` which is subdirectory of the `LintDir` directory.

6. Repeat the process of creating a global macro for each of the following. Ensure any file extensions (such as `.exe` or `.lnt`) are included in the RHS of the macro definition:
  - (a) `LintExecutable=<name of your PC-lint Plus executable>`
    - i. This macro should not include the path to the executable.
  - (b) `LintConfiguration=<name of your .lnt config file>`
    - i. Here, the `.lnt` config file would be one of: `co-segger-cc1.lnt`, `co-segger-cc1plus.lnt`, or `co-segger-cc.lnt` if you have adhered to the naming conventions for `--config-output-lnt-file` presented in [2.3.11](#). However, these names can be custom. If you have used a custom name, `LintConfiguration` should expand to that name.
7. Click "OK" to close the global macros menu.
8. Click "OK" to close the options window.
9. Next, open `tools.xml` by selecting "File", then "Open Studio Folder", then "External Tools Configuration". The file will open in the SEGGER editor.

10. Copy and paste the following XML entry before the line in `tools.xml` that contains `</tools>`:

```
<!-- PC-lint Plus - https://pclintplus.com/ -->
<item name="Tool.PCintPlus">
  <menu>&PC-lint Plus</menu>
  <text>PC-lint Plus</text>
  <tip>Use PC-lint Plus to analyze the selected file or folder</tip>
  <key>Ctrl+L, Ctrl+P</key>
  <match>*.c;*.cpp</match>
  <message>Analyzing</message>
  <commands>
    &quot;$(LintDir)/$(LintExecutable)&quot; -i&quot;$(LintDir)/lnt&quot;;
    &quot;$(LintDir)/lnt/$(LintConfiguration)&quot; &quot;$(DEFINES)&quot;;
    $(INCLUDES) -u -b +ffn -width(0,4) -hF1
    &quot;-format_category[info,Info]&quot;;
    &quot;-format_category[supplemental,note: supplemental]&quot;;
    &quot;-format=%f:%l:%C:\s%t:\s%m [-e%n]&quot;; &quot;$(InputPath)&quot;;
  </commands>
</item>
```

11. This XML entry assumes that your C source files have the extension `.c` and your C++ source files have the extension `.cpp`. If you are using different extensions, you will need to modify the `match` element.
12. Ensure the entire `commands` section is on a single line with a space between each option.
13. Finally, save `tools.xml` and restart SEGGER IDE.

To run PC-lint Plus on a single file, right click the desired source file, then select "PC-lint Plus".

To run PC-lint Plus on the entire project, right click the desired source directory, then select "PC-lint Plus". Alternatively, `Ctrl + L + P` can be used to run PC-lint Plus. Ensure that you have the desired source file or source directory highlighted in the Project Explorer before using this hotkey.

### 2.3.21 Using the `pclpvscfg.exe` GUI utility to generate a compiler and project configuration for Microsoft Visual Studio

PC-lint Plus ships with a GUI utility to automate the steps involved in generating simple compiler and project configurations for Microsoft Visual Studio. It will generate a batch file that can then be executed to generate a compiler and project configuration. `pclpvscfg.exe` can be found in the `config/` directory of the PC-lint Plus distribution. This utility also relies on `vswhere.exe` which is redistributed with permission from Microsoft. `pclpvscfg.exe` must be executed from the same directory as `vswhere.exe` and `pclp_config.py` must be in the same directory as `compilers.yaml` and `imposter.c` (as they are in the distribution).

Using `pclpvscfg.exe`:

1. If your project or solution builds a 64-bit target, check the "64-bit build" checkbox.
2. The main dialog lists detected Visual Studio installations. Double click a Visual Studio installation path from the list to continue. This must be the correct Visual Studio version for your project or solution.
3. To use the generated batch file to create a configuration: *from a command prompt*, execute the saved batch file. Note that this batch file will clean and rebuild the selected project or solution. *Cleaning a build will delete build files*. If no errors occur when running the generated batch file, the configuration files will be ready for use in the locations chosen in `pclpvscfg.exe`.

Troubleshooting:

- If a Python installation could not be detected by the availability of `python` in your `%PATH%`, a warning will appear. The batch file that the utility will generate requires `python` to be available.

- If a Python installation was detected but a necessary package could not be detected, a warning will appear indicating either that the package appeared not to be installed or that the installation status could not be determined. Ensure the package is installed before using the batch file that the utility will generate.
- If an error appears to have occurred after the end of your build, for example at the linking stage, or if compilation continued after the error, the configuration files may still be ready for use. Only an error which stops the build before all files have been passed to the compiler will interfere with the configuration process.
- If you receive an error that "build tools cannot be found" this often means the selected Visual Studio installation path does not match the project or solution.
- If the compiler fails to execute or you encounter a build configuration error, ensure that the "64-bit build" checkbox was set correctly.
- If your solution was generated by CMake you may need to e.g. copy the "Win32" configuration to "x86" or manually add a configuration option to `msbuild` in the generated batch file.
- If you encounter syntax errors related to the macro `_CRT_BEGIN_C_HEADER` in Visual Studio standard library headers when running PC-lint Plus with a Visual Studio project configuration then module was likely processed without a Visual Studio compiler configuration. Ensure that the compiler configuration is present as an argument to PC-lint Plus prior to the project configuration.
- If you receive an error about 'utf8' such as `UnicodeDecodeError: 'utf8' codec can't decode` when running the generated batch file from the Windows command line then you may need to change the code page of that session to UTF-8 using the command `chcp 65001` before running the generated batch file.

### 2.3.22 `pclp_config` Options Reference

`pclp_config` accepts a variety of options that can be used to fine-tune generated configurations and for troubleshooting purposes. The most common options used with `pclp_config` are described above, below is a complete list of options supported by `pclp_config`. Options that accept arguments can be specified as either `--option=arg` or `--option arg`.

Option	Description
<code>--help</code>	Show the help screen.
<code>--compiler-database</code>	Used to specify the location of the compiler database; <code>compilers.yaml</code> is the default.
<code>--list-compilers</code>	Dump a list of compilers supported by the compiler database along with a short description of each.
<code>--compiler</code>	Used to specify the <i>name</i> of a compiler or compiler family from the list of supported compilers.
<code>--compiler-bin</code>	Used to specify the full path of the compiler binary for operations that need access to the compiler (such as the <code>--generate-compiler-config</code> and <code>--compiler-version</code> operations).

<code>--compiler-version</code>	Runs the specified compiler and attempts to extract and display version information. Useful for troubleshooting. Emits 'None' if extraction fails for any reason. Must be used with <code>--compiler</code> and <code>--compiler-bin</code> .
<code>--generate-compiler-config</code>	Instructs <code>pclp_config</code> to generate a compiler configuration for the specified compiler. Usually used with <code>--compiler</code> and <code>--compiler-bin</code> .
<code>--generate-project-config</code>	Instructs <code>pclp_config</code> to generate a project configuration using compile commands logged from running the build process with the <i>imposter</i> program in place of the compiler. Used with the <code>--compiler</code> , <code>--imposter-file</code> , and <code>--config-output-lnt-file</code> options.
<code>--config-output-lnt-file</code>	Specifies the name of the configuration file to write when generating compiler or project configurations.
<code>--config-output-header-file</code>	Specifies the name of the compiler configuration header file to generate.
<code>--imposter-file</code>	Specifies the name of the log file containing compile commands logged by the <i>imposter</i> process.
<code>--compilation-db</code>	Specifies the name of the JSON compilation database used to generate a project configuration.
<code>--prefix-directory-options</code>	Specifies additional compiler options for which the argument should be prefixed by the directory entry in a JSON compilation database if the argument is not an absolute path. By default, the options <code>-I</code> , <code>/I</code> , and <code>@</code> receive this treatment.
<code>--response-file-prefix</code>	Specifies the string used to introduce the name of a response file within a JSON compilation database. The default value is ' <code>@</code> ', if your compiler uses a different option or prefix to specify a response file, e.g. <code>-f</code> , this option can be used recognize it.
<code>--posix-command-parsing</code>	Specifies that posix-like parsing behavior should be employed when processing response files, arguments embedded in the command field of a JSON compilation database, and in the application of the <code>--shell-parse-compiler-options</code> option. In particular, backslashes are treated as escape characters and quotes may be used to specify arguments containing spaces. This is the default behavior on Linux and macOS.
<code>--no-posix-command-parsing</code>	Specifies that non-posix-like parsing behavior should be employed when processing response files, arguments embedded in the command field of a JSON compilation database, and in the application of the <code>--shell-parse-compiler-options</code> option. In particular, backslashes and quotes are treated as literal characters. This is the default behavior on Windows.

<code>--shell-parse-compiler-options</code>	Specifies that shell-like parsing should be employed when processing arguments to <code>--compiler-options</code> , <code>--compiler-c-options</code> , and <code>--compiler-cpp-options</code> . The <code>--posix-command-parsing</code> and <code>--no-posix-command-parsing</code> options control whether this uses posix-like or non-posix-like parsing.
<code>--source-pattern</code>	A regular expression used to distinguish source modules from options in the compile commands log generated by the imposter process. The default value is <code>.*\.(c cpp)\$</code> and matches any string that ends with <code>.c</code> or <code>.cpp</code> .
<code>--module-include-pattern</code>	A regular expression pattern used to specify which modules will be included in a generated project configuration. This option may be used multiple times to specify multiple patterns, a module that matches any of the provided patterns will be included unless it also matches a pattern specified by <code>--module-exclude-pattern</code> .
<code>--module-exclude-pattern</code>	A regular expression pattern used to specify modules that should be excluded from a project configuration. This option may be used multiple times to specify multiple patterns. A module matching any of the patterns provided by this option will be excluded from the generated project configuration.
<code>--compiler-options</code>	A space separated list of base compiler options. If you are targeting an architecture other than the compiler's default, you should include the option that specifies the target architecture when generating a compiler configuration to ensure the correct values for size options in the generated configuration. Compiler options specified with this option are applied when the compiler is invoked in either C or C++ mode.
<code>--compiler-c-options</code>	Similar to <code>--compiler-options</code> but only used when invoking the compiler in C mode. Use for C-only language options such as setting the language version.
<code>--compiler-cpp-options</code>	Similar to <code>--compiler-options</code> but only used when invoking the compiler in C++ mode. Use for C++-only language options such as setting the language version.
<code>--repl</code>	Start a Read Eval Print Loop where compiler options are read from stdin and transformed PC-lint Plus options are printed to stdout. Requires <code>--compiler</code> . This is a debugging option.
<code>--scavenge-files</code>	Specifies the list of files to attempt to extract macro names from when performing macro scavenging.
<code>--scavenge-dirs</code>	Specifies the directories to recurse looking for files to extract macro information from when performing macro scavenging.
<code>--scavenge-pattern</code>	A regular expression specifying files that should be examined for macro information when using <code>--scavenge-dirs</code> .

<code>--dont-filter-feature-test-macros</code>	Instructs <code>pclp_config</code> to not filter C++ feature test macros in the generated compiler configuration header file. By default, feature test macros extracted from the compiler which would be limited or removed to match the language support in PC-lint Plus.
<code>--header-option-use-enclosing-directory</code>	Use the built-in <code>%ENCLOSING_DIRECTORY%</code> environment variable to provide an ‘absolute’ path for the compiler configuration <code>-header</code> option.
<code>--debug-dont-show-compiler-errors</code>	When an unexpected condition is encountered, do not show the associated compiler invocation command line, standard input, standard output, and standard error.
<code>--debug-show-command</code>	Show the compiler invocation command lines.
<code>--debug-show-input</code>	Show the compiler invocation standard input.
<code>--debug-show-output</code>	Show the compiler invocation standard output and error.
<code>--debug-show-trace</code>	Show the internal processing steps.
<code>--debug-show-tempfile-cleanup</code>	Show debugging messages regarding the cleanup of generated temporary files.
<code>--debug-show-tempfile-derived</code>	Show debugging messages regarding the <code>tempfile_derived</code> rules.



### 2.3.23 imposter Options Reference

As previously mentioned, the `imposter` program uses environment variables to control its behavior instead of traditional command line arguments. The most commonly used variables are `IMPOSTER_LOG` and `IMPOSTER_COMPILER` although the imposter supports a variety of other options for specific situations. A full list of options supported is provided below.

Environment Variable	Description
<code>IMPOSTER_EXIT_SUCCESS</code>	The value with which the imposter should exit when not invoking a compiler and when no error occurs. This should be the same value your compiler exits with when there is no compilation error or the value your build system expects for a successful compilation. If no success value is supplied, 0 is used.
<code>IMPOSTER_EXIT_FAILURE</code>	The exit code used when an error is encountered before invoking the compiler. This value is used if the log file cannot be written to, if we run out of memory, or if the compiler fails to invoke. If the compiler is successfully invoked, the imposter returns the value that the compiler returned. If no failure value is supplied, the default of 1 is used.
<code>IMPOSTER_LOG</code>	The name of the log file to which compiler commands should be written. If an absolute path is not provided, the path is relative the directory in which the imposter is invoked. If no value is provided, output is written to stdout.
<code>IMPOSTER_COMPILER</code>	The full path of the compiler to invoke after logging invocation information. If no value is provided, no compiler is invoked (unless <code>IMPOSTER_COMPILER_ARG1</code> is set as described below) and the imposter exits with <code>IMPOSTER_EXIT_SUCCESS</code> on success.
<code>IMPOSTER_COMPILER_ARG1</code>	If this variable is set to any value (including an empty value) and <code>IMPOSTER_COMPILER</code> is NOT set, the compiler is expected to be provided as the first argument to the imposter program and will be executed with the remaining argument list. If this variable is set and there is no first argument, the imposter will exit with the failure exit code. This functionality is useful when the imposter needs to stand in for multiple compilers during the build process, such as both a C and C++ compiler.
<code>IMPOSTER_AUTO_RSP_FILE</code>	If this variable is set, the value is interpreted to be a response file that should be processed and logged to the beginning of the invocation.
<code>IMPOSTER_NO_RSP_EXPAND</code>	If an argument is received that starts with '@', this is treated as a response file and the argument is replaced with the parsed arguments contained within the response file (the name left after removing the '@'). If the file cannot be opened, or <code>IMPOSTER_NO_RSP_EXPAND</code> is set, the entire argument is logged as is. Note that in all cases, if the compiler is invoked, it receives the unexpanded argument; the expansion is limited to the logging process.

**IMPOSTER\_PRE\_2008\_PARAMS**

When parsing response files, the parameters it contains are processed using the Windows command line parameter parsing rules. There is a subtle and undocumented difference between the way handling of consecutive quotes was handled prior to 2008 and after 2008. By default, the post-2008 rules are employed. If this variable is set, the pre-2008 rules are instead employed.

**IMPOSTER\_INCLUDE\_VARS**

A semi-colon separated list of environment variables from which to extract header include information. If this variable is not set, the default list of: `INCLUDE;CPATH;C_INCLUDE_PATH;CPLUS_INCLUDE_PATH` is used. Each environment variable processed is expected to contain a list of paths, which are emitted as `-I` options in the log file. The delimiter used and how to change it are described below in **IMPOSTER\_INCLUDE\_DELIM**. Environment variables are processed in the order provided. Environment variables included in this list that are not set are ignored. Setting **IMPOSTER\_INCLUDE\_VARS** to an empty value will disable this processing.

**IMPOSTER\_INCLUDE\_DELIM**

The character(s) recognized as delimiters when parsing include directories specified from environment variables (such as those specified by **IMPOSTER\_INCLUDE\_VARS**). By default, this is `'` on Windows and `:` on other platforms. Setting this variable will override the default. Each character appearing in the value of this variable will be considered to be a delimiter. For example, setting this variable to `'|!:'` will cause any of these characters to separate directories appearing in the include variables.

**IMPOSTER\_MODULES\_IN\_WORKING\_DIR**

If this variable is set, the working directory will be prepended to each argument which appears to be the relative path of a module. Apparent options, absolute paths, or arguments lacking one of the extensions defined in **IMPOSTER\_MODULE\_EXTENSIONS** will be ignored.

**IMPOSTER\_MODULE\_EXTENSIONS**

If this variable is set, its value replaces the default module extension list: `".c;.C;.cpp;.CPP;.cc;.CC"`. The value is a case-sensitive list of extensions (including the dot) delimited by semicolons. It is used only to determine what constitutes a module for the purposes of prepending the working directory and will not otherwise affect the configuration process.

**IMPOSTER\_PATH\_ARGUMENT\_RELATIVE\_TO  
\_WORKING\_DIR\_OPTION\_INTRODUCERS**

If this variable is set, any argument that initially matches one of the provided semicolon-delimited strings will have the working directory inserted after the option introducer if the remainder of the option does not appear to be an absolute path. If an option introducer from this list appears as a standalone compiler argument with no appended suffix to insert a working directory before, then the working directory will be inserted before the subsequent argument (if any) unless the next argument is an absolute path or an option. This is intended for options like a `-I` option where, in e.g. `-Irelative/path` the working directory needs to be inserted just after `-I` and in `-I path/as/next/arg` it needs to be added to the following argument. By contrast, a `-Dmacro=value` should not have the working directory inserted before the macro name.

#### 2.3.24 Installing `pclp_config` prerequisites

`pclp_config` is implemented in Python and requires the Python runtime and certain modules to be installed before it can be used. Installing these dependencies is a simple process described below.

#### 2.3.25 Installing python

The `pclp_config` program requires Python 2.7 or higher (3.5 or higher is recommended). This section will walk you through the simple process of installing Python and the modules required by `pclp_config`.

- Windows  
Download and run the latest [Python](#) release for Windows.
- Linux  
Python is probably already installed; try running `python --version` to check. If it isn't, you can use a package manager to install it:
  - `sudo apt-get install python` or `sudo apt-get install python3` on Ubuntu
  - `sudo yum install python` on RedHat or Fedora

You can also install python from source:

- Download the python sources, uncompress, and run `./configure && make altinstall`.

See [Using Python on Unix platforms](#) for more information.

- macOS  
Python is installed by default on macOS.
- FreeBSD  
Python is probably already installed; try running `python --version` to check. If it isn't, you can install it with the command `pkg_add -r python` or build it from source.  
See [Using Python on Unix platforms](#) for more information.

#### 2.3.26 Installing required modules

The `pclp_config` program uses two modules that are not part of the standard python distribution: `regex` and `yaml`. The easiest way to install python modules is with the `pip` module. These modules can be installed from the command line as follows:

- `python -m pip install regex`
- `python -m pip install pyyaml`

Recent versions of Python (2.7.9 and up) already contain `pip`. If `pip` is not installed, you can install it using:

- On macOS: open a terminal window and run: `sudo easy_install pip`
- On other platforms: Download the [get-pip.py](#) script and run it from a terminal window with `python get-pip.py`.

## 2.4 Configuring manually

**Note:** PC-lint Plus ships with an automated configuration tool named `pclp_config.py`, which can be found in the `config/` directory of the PC-lint Plus distribution. This Python script simplifies the process of configuring PC-lint Plus for your compiler and project. We recommend the use of `pclp_config.py` over manual configuration for all compilers supported by `pclp_config.py`. See section 2.3.2 to get started with `pclp_config.py`.

### Step 1 - Configure Include Directories using the `-i` option

PC-lint Plus needs to know where the compiler looks for system headers, there are several ways to obtain this information depending on the capabilities of the compiler including:

- The preprocessor can be used to emit information about the location of included files. A single-line C source module that just includes a system header (e.g. `#include <stdio.h>`) can be passed through the compiler in preprocessor only mode in an attempt to obtain the location of the header. The preprocessed output often includes the full paths of included files via `#line` directives, which can be used to determine the locations of header files. For example, if you are programming on Windows, create a C file called `xx.c` that contains only `#include <stdio.h>`.

Then, from a Developer Command Prompt use the command:

```
cl /P xx.c
```

The `xx.i` will contain line directives that will indicate the directory containing `stdio.h`

- If the compiler provides an option to list system include paths, this option can be employed. For gcc and clang based compilers, the options `-v -xc -E /dev/null` will emit information that includes the system include search path for C headers. For C++ include paths, the corresponding options are `g++ -v -xc++ -E /dev/null`. For other compilers, consult the compiler's documentation to determine if such an option exists.
- An intentional compile-time error can be introduced in such a way to elicit a message from the standard header, the error message would presumably include the path information. For example, the following program:

```
#define fprintf @
#include <stdio.h>
```

produces the following error when processed by clang:

```
/usr/include/stdio.h:356:12: error: expected identifier or '('
extern int fprintf (FILE *__restrict __stream,
                  ^
```

which includes the full path of `stdio.h` (`/usr/include`) in the output.

- A filesystem search of the standard library headers can be employed. A search for e.g. `stdio.h` can be used to determine where the header files reside. The downside of this approach is that if there are several standard library implementations installed, it may not be obvious which one is used by a particular compiler.

Once the list of standard library search paths has been determined using one of the above methods, create a file named `lint-includes.lnt` containing this list with each directory prepended with `-i`, for example:

```
-i/usr/lib/gcc/x86_64-linux-gnu/4.8/include
-i/usr/local/include
-i/usr/include/x86_64-linux-gnu
-i/usr/include
```

If you have directories that contain space characters, you will need to surround the path with double quote characters, e.g.:

```
-i"/usr/include"
```

Header files found while searching directories specified by `-i` options are treated as library by default. If the option `+libclass()` is used or `+libclass(angle)` is used and the headers are included without using angle brackets, headers found in these directories will not be considered library. Corresponding `+libdir` options can be added to `lint-includes.lnt` to ensure that these directories are treated as library in such cases, e.g.:

```
+libdir(/usr/lib/gcc/x86_64-linux-gnu/4.8/include)
```

## Step 2 - Extract Predefined Macros

PC-lint Plus needs to know about compiler-defined macros since these will be used by the implementation. Many compilers provide an option to dump such predefined macros to a file. This is by far the easiest way to extract the macros if your compiler supports it.

- **Method 1 - Extract macros using the compiler**

Below is a table that documents the invocation that can be used to dump either C or C++ macros. If you are using one of these compilers, or a compiler that is a derivative of one of these, you can use the corresponding invocation to dump the macros. Otherwise, consult your compiler's documentation for the equivalent options.

Compiler	C Command	C++ Command
clang	<code>clang -dM -E -xc /dev/null</code>	<code>clang++ -dM -E -xc++ /dev/null</code>
GCC	<code>gcc -dM -E -xc /dev/null</code>	<code>g++ -dM -E -xc++ /dev/null</code>
IBM XL	<code>xlc -qshowmacros -E /dev/null</code>	<code>xlc++ -qshowmacros -E /dev/null</code>
Solaris Studio	<code>cc -xdumpmacros -E /dev/null</code>	<code>CC -xdumpmacros -E /dev/null</code>

- **Method 2 - Extract enumerated macros using the preprocessor**

If your compiler does not support dumping pre-defined macros then you will need to create a source file listing pre-defined macro names and compile this file to obtain their values. For example, compiling:

```
#define S2(x) #x
#define S1(x) S2(x)
#define DV(x) "#define " #x " " S1(x)

#if defined(_MSC_VER)
    #pragma message(DV(_MSC_VER))
#endif

#if defined(_MSC_FULL_VER)
    #pragma message(DV(_MSC_FULL_VER))
#endif
```

with one version of the `cl.exe` compiler produces the output:

```
#define _MSC_VER 1924
#define _MSC_FULL_VER 192428316
```

By convention, C language macros should be placed into a file named `lint_cmac.h`, C++ language macros should be placed into a file named `lint_cppmac.h`.

**Step 3 - Establish the sizes of the fundamental types**

PC-lint Plus needs to know the size of the basic types (`int`, `short`, `float`, `double`, `pointers`, etc.). In this step we'll create a file called `size-options.lnt` that contains the sizes of these types. To generate this file, compile and run the following C program using the same compiler and target options used to compile the code that PC-lint Plus will be processing.

```
#include <stdio.h>
#include <wchar.h>
int main(void) {
    printf("-ss%zu  ", sizeof(short));
    printf("-si%zu  ", sizeof(int));
    printf("-sl%zu  ", sizeof(long));
    printf("-sll%zu ", sizeof(long long));
    printf("-sf%zu  ", sizeof(float));
    printf("-sd%zu  ", sizeof(double));
    printf("-sld%zu ", sizeof(long double));
    printf("-sp%zu  ", sizeof(void*));
    printf("-sw%zu\n", sizeof(wchar_t));
}
```

Running this program will generate a line that looks something like this:

```
-ss2 -si4 -sl8 -sll8 -sf4 -sd8 -sld16 -sp8 -sw4
```

Copy and paste this line, or redirect the output of the program, to a file called `size-options.lnt`

If you are cross-compiling and cannot execute a binary compiled for the target to obtain this output then you can consult your compiler and target platform documentation for a table of type sizes. Alternatively, type sizes can be obtained through compiler error messages by defining arrays which are illegally of length zero in the presence of a particular type size, e.g. attempting to compile:

```
int si8[8 != sizeof(int)];
int si4[4 != sizeof(int)];
int si2[2 != sizeof(int)];

int ss8[8 != sizeof(short)];
int ss4[4 != sizeof(short)];
int ss2[2 != sizeof(short)];

int sl8[8 != sizeof(long)];
int sl4[4 != sizeof(long)];
int sl2[2 != sizeof(long)];

int sll8[8 != sizeof(long long)];
int sll4[4 != sizeof(long long)];
int sll2[2 != sizeof(long long)];

int sf8[8 != sizeof(float)];
int sf4[4 != sizeof(float)];
int sf2[2 != sizeof(float)];

int sd8[8 != sizeof(double)];
int sd4[4 != sizeof(double)];
int sd2[2 != sizeof(double)];

int sld8[8 != sizeof(long double)];
```

```

int sld4[4 != sizeof(long double)];
int sld2[2 != sizeof(long double)];

int sp8[8 != sizeof(void*)];
int sp4[4 != sizeof(void*)];
int sp2[2 != sizeof(void*)];

int sw8[8 != sizeof(L'a')];
int sw4[4 != sizeof(L'a')];
int sw2[2 != sizeof(L'a')];

```

should trigger a compiler error on one line out of each group of three lines. The name of each array that triggers an array of length zero error can be used as a size option after prepending it with a single dash.

#### Step 4 - Enable compiler-specific extensions

Most compilers implement a number of extensions to the C and C++ languages, such as GCC's Case Ranges or Microsoft's Assembly blocks. Embedded compilers often support additional language extensions. If these extensions are used by projects that will be processed by PC-lint Plus, they should be enabled in the configuration. Because there is a wide range of extensions supported by various compilers, we provide flexible compiler adaptation options. See section [4.12.1 Customization Facilities](#) and section [4.11 Flag Options](#). Contact support if you run into issues configuring PC-lint Plus for your compiler.

#### Putting it all Together

If you have followed the steps outlined above to manually create a configuration for your compiler, you should now have the following:

- `lint-includes.lnt` contains the system include paths.
- `lint_cmac.h` and/or `lint_cppmac.h` contains C and C++ compiler macro definitions.
- `size-options.lnt` contains options describing the sizes of the fundamental data types.
- Your chosen compiler adaptation options.

Now it's time to put it all together in one place. Create a file named `std.lnt` with the following contents:

```

lint-includes.lnt
size-options.lnt
+libh(lint_cmac.h) -header(lint_cmac.h) // and/or lint_cppmac.h as appropriate
// <chosen compiler adaptation options for compiler extensions>

```

## 3 The Command Line

### 3.1 Indirect (.lnt) files

If the extension is `.lnt` or equivalent (see the `+lnt` option), the file is processed as an indirection, in which case it may contain one or more lines of information that would otherwise be placed on the command line. Indirect files may reference other indirect files and may be nested to any depth. Indirect files and other files may be interspersed in any manner desired. Indirect files may contain comments in addition to options and files. Both the standard C-style comment `/*...*/` and the C++ style comment `// . . . end-of-line` are supported. Comments following options should be separated by at least one space character to prevent the comment from being processed as part of the option.

Environment variables are expanded inside indirect files when surrounded by `'%'` characters. Thus an indirect file containing:

```
%SOURCE%/a.c    // first file
%SOURCE%/b.c    // second file
```

employs the environment variable `SOURCE` to specify where the files are located. The environment variable specification is case sensitive.

If an indirect file is not found in the current directory, a search is made in the usual places. For example, if `lin.bat` contains:

```
C:/lintpp/pclp64 -iC:/lintpp std.lnt %*
```

The `std.lnt` will be found in the directory `C:/lintpp` (if not overridden by the existence of `std.lnt` in the current directory).

`std.lnt` might contain:

```
co.lnt
options.lnt
```

This illustrates the nesting of indirect files.

### 3.2 Exit Code

The operating system supports the notion of an exit code whereby a program may report a byte of information back to a controlling program. By default, PC-lint Plus will return 0 if processing is completed without any fatal or internal errors and an exit code of 1 otherwise. If the `frz` flag is turned OFF (such as with `-frz`) then the exit code will default to the number of messages generated (with an upper bound of 255) and the options `-exitcode`, `-zero`, and `-zero_err/+zero_err` can be used to manipulate the exit code.

If run from a shell, the return value can be obtained after PC-lint Plus terminates using `echo $?` for the Bash shell for Linux and macOS, `echo %ErrorLevel%` for `cmd.exe` on Windows, or `echo $LastExitCode` on PowerShell.

### 3.3 Built-in version environment variables

The following "built-in" environment variables are expanded to the corresponding version information when appearing in an option and surrounded by percent symbols.

- `LINT_MAJOR_VERSION`  
The major version number, e.g. 1.
- `LINT_MINOR_VERSION`  
The minor version number, e.g. 0.



- `LINT_PATCH_LEVEL`  
The patch level, e.g. 1.
- `LINT_VERSION`  
The full version number incorporating the previous three components, using two digits for the minor version as described in Section [18.1 Preprocessor Symbols](#), e.g. 1001.
- `LINT_BETA_LEVEL`  
For a beta release, a number representing the beta level, e.g. 1. For non-beta releases, this variable has the value 0.

These environment variables can be used to conditionally execute PC-lint Plus options. For example:

```
-cond(%LINT_VERSION% >= 1002,true-options,false-options)
```

## 4 Options

### 4.1 Rules for Specifying Options

Options begin with a plus (+) or minus (-), except as for the special **!e** option. Strings that start with any other character are presumed to be filename arguments.<sup>2</sup>

Options are processed **in order** meaning that an option specified after a filename will not take effect until after that file is processed. The effect of options encountered within a source file are limited to the file in which they appear.

Options may be provided on the command line, via the LINT environment variable, within of indirect files, and within lint comments. In all cases, option parsing follows the same general rules although command line arguments may be subject to shell interpolation before they reach PC-lint Plus so additional quoting may be required for options that include characters that your shell considers special.

#### 4.1.1 Options within Comments

Options may be placed within a source code file embedded within comments having the form:

```
/*lint option1 option2 ... optional-commentary */
```

or

```
//lint option1 option2 ... optional-commentary
```

Options within comments should be space-separated. The *optional-commentary* should, of course, not begin with a plus, minus, or exclamation point. Note that the 'lint' within the comment must be lower-case as shown and must lie immediately adjacent to the '/\*' or '//'. The */\*lint* comment may span multiple lines. Note that within indirect files (*.lint* files), options need not and should not be placed within a lint comment.

#### 4.1.2 Lint Comments inside of Macro Definitions

Lint comments may appear in the definition of a macro. Lint comments of the form */\*lint ... \*/*, when written as part of a macro definition, are not processed immediately but instead are retained and expanded when the macro is used. Lint comments of the form *//lint ...* are not retained as part of the macro definition and are processed like any other lint comment appearing outside a macro. For example:

```
#define UNUSED
```

will result in message 750 (local macro not referenced) if the macro is not used. One way to suppress this message is:

```
#define UNUSED //lint !e750
```

The seemingly equivalent:

```
#define UNUSED /*lint !e750*/
```

will not work because the C-style lint comment is not processed until the macro is expanded.

Function-like macro arguments are expanded within C-style lint comments appearing inside the macro. For example:

---

<sup>2</sup>To specify the name of a file that begins with -, +, or ! you can either provide a relative or absolute path that does not begin with one of these characters (e.g. *./-funnyfile*) or enclose the name in quotes (e.g. *"-funnyfile"*).

```
#define SuppressInBlock(...) /*lint --e{__VA_ARGS__} */

void foo() {
    SuppressInBlock(438, 529)
    int i = 0;
}
```

will suppress messages 438 and 529 within the body of `foo`. The `-p` option can be used to see how the macro is expanded:

```
void foo() {
    /*lint --e{438, 529} */
    int i = 0;
}
```

#### 4.1.3 Options on the Command Line

Options specified on the command line are parsed until the end of the command-line argument, even if a space occurring earlier would normally signify the end of the option. This is to prevent unexpected behavior from options that look properly quoted on the command-line but are provided to PC-lint Plus without quotes due to shell interpolation. For example, the option `-"format=%(%f %l %) %t %n: %m"` is a valid option but when provided on the command line, some shells may eat the quotes such that PC-lint Plus sees only `-format=%(%f %l %)%t %n: %m` without the quotes. Because options are separated by spaces, in other contexts this would be interpreted as 5 options (the first being `-format=%(%f)`), none of which are valid. Since multiple options are not allowed in one command-line argument, PC-lint Plus assumes this is intended to be a single option and will parse it as such.

Shell interpolation can cause other problems that cannot be rectified by PC-lint Plus so it is important to understand your shell's handling of quoting characters when using non-trivial command-line arguments. You should consider placing your options within indirect (`.lint`) files where they will not be subject to shell transfiguration.

#### 4.1.4 Specifying Option Arguments

Some options take a single argument using the `option=arg` syntax. Options are separated by whitespace so there must not be any unquoted spaces around the `=` sign or within the value. Multiple arguments options use `option(arg1 ,arg2 , ...)`. In this form, the option is immediately followed by a parenthesized list of comma-separated arguments. In place of parentheses, curly braces or square brackets may be used to delimit the argument list. In place of the comma, an exclamation point (!) may be used to separate option arguments. In the `option=arg` form, the comma and exclamation symbols do not have any special meaning.

#### 4.1.5 Quoted Options and Arguments

Arguments can be quoted by surrounding the entire argument in double-quotes. Quoting is the only way that arguments may contain whitespace when using the `option=arg` form (for which a space would normally signify the end of the option). For example:

```
-message="This is a test"
```

will emit the message:

```
"This is a test"
```

(including the quote characters). To obtain the same result without the quote characters being emitted, place the initial quote before the `=`, e.g.:

```
-message="=This is a test"
```

will emit the message:

```
This is a test
```

The initial quote can be placed anywhere between the '-' and the '=' to achieve the same effect.

When using an argument list, quoted arguments also suppress the meaning of brace characters and argument separators. For example:

```
-message( ",!)) )" )
```

will emit the message:

```
,!)) )
```

Note that for quoted arguments in parenthesized argument lists, the surrounding quotes are not preserved. For unquoted arguments, the leading and trailing whitespace is removed but whitespace within the argument is preserved. Leading and trailing whitespace is preserved in quoted arguments.

#### 4.1.6 Quotes in Arguments

Quotes (") have no special meaning when appearing inside of an unquoted option, e.g. they do not introduce a quoted region and do not have to be paired. For example:

```
-esym(714, operator "" _x)
```

Suppresses message 714 for the literal operator function associated with the user-literal extension \_x. Because the quotes occur in the middle of an unquoted argument, they are taken literally.

Quotes present in quoted arguments are treated as literal quotes as long as they cannot be mistaken for the closing quote of an argument. For example, in the option:

```
-esym(714, "operator "" _x")
```

The two inner quote characters are taken to be literal quotes because what follows is the continuation of the argument. In general, quotes inside of a quoted field are taken as literal unless the next non-whitespace character is a , or !, or is the closing brace character that terminates the parenthesized argument list.

#### 4.1.7 Braces within Argument Lists

The brace characters (, [, {, }, ], and ) must be balanced within a non-quoted argument appearing in a parenthesized argument list. Within an unbalanced region, argument separator characters and unbalanced closing braces are ignored and the brace-enclosed list cannot be terminated. For example:

```
-message({ this ) does not end the option nor this , the argument })
```

Prints the text:

```
{ this ) does not end the option nor this , the argument }
```

Within the unbalanced braced region between the { and }, the , and ) characters do not have any special meaning. Note that the braces that introduced a balanced region are preserved. This balancing is necessary to support options such as:

```
-function(operator new(r))
```

#### 4.1.8 Braces and Quotes

Balanced sequences are not recognized within a quoted region, in other words brace characters have no special meaning in a quoted argument. For example, in:

```
-message( " ) [ " )
```

the ) in the quoted string does not end the option, neither does the [ introduce a new balanced region.

### 4.1.9 Brace Types in Brace-Enclosed Argument Lists

Any of the `(`, `{`, or `[` may be used to start an argument list that will be terminated at the first occurrence of a corresponding unquoted closing brace character that occurs in a balanced region. For example, `-message(test)`, `-message{test}`, and `-message[test]` are all equivalent.

### 4.1.10 Option Display

Because options can be placed in obscure places and then forgotten, the verbosity option `-vo` can be used to display all options as they are encountered.

### 4.1.11 Expansion of Environment Variables in Options

Environment variables are expanded when surrounded by percent signs (`%`), this happens in options and arguments as well as in filenames. This expansion occurs even inside of quoted option strings and braced lists but only when the text between the percent signs corresponds to the name of a defined environment variable.

Environment variables are expanded before options are parsed, which means that the expansion of an environment variable could contain part of an option or even multiple options.

Note that because the expansion of environment variables occurs before the options are processed, a `-setenv` option will not affect following options in the same line comment. Similarly, a `-setenv` option appearing in a configuration file will not affect environment variables within the same file (as the environment variables will have been expanded before any `-setenv` options are processed). In general, we recommend that environment variables be defined at the beginning of processing and not be changed afterwards.

Expansion of environment variables can be disabled by setting the `fee` flag option to OFF.

The dynamic built-in environment variable `%ENCLOSING_DIRECTORY%` expands to the absolute path of the enclosing directory of the configuration file, module, or header in which the option appears. Outside of a file, it expands to the current working directory. When using this environment variable as the initial component of larger pathname, we recommend using a forward slash as the directory separator even on Windows. For example, the include option `-i%ENCLOSING_DIRECTORY%/library` could be placed in a configuration file to add a directory called `library` present in the same directory as the configuration file to the include search list.

### 4.1.12 Escaping Special Characters

The characters `{}`, `()`, `[], !` have special meanings within options. If the `fbe` flag is ON, `\` (backslash) becomes a special character which disables the special meaning of a subsequent special character which must immediately follow it. When the flag is OFF, the backslash character is interpreted literally.

Note that only option parsing is affected, not option-specific handling of arguments. Even when the flag is ON, a backslash cannot be used to “escape” the meaning of characters significant only to the interpretation of an option argument, e.g. backslash cannot be used to inhibit wildcards in a suppression option.

## 4.2 Option Reference

The table below summarizes the available PC-lint Plus options. Options that were introduced in PC-lint 9 or earlier are marked — in the version column.

Option	Summary	Version
<code>?</code>	displays help	—
<code>-I</code>	add search <i>directory</i> for <code>#include</code> directives	—
<code>-\$</code>	permit <code>\$</code> in identifiers	—
<code>-a</code>	set the alignment of various types	—
<code>-append</code>	append <i>String</i> to diagnostic <code>#</code> when issued	—
<code>-ar_limit</code>	set the operator arrow depth limit to <i>n</i>	2.0
<code>-astquery</code>	register a Query to be evaluated during AST traversal	2.0
<code>+b</code>	redirect banner output to stdout	—
<code>++b</code>	produce banner line	—
<code>-b</code>	suppress banner output	—
<code>-br_limit</code>	set the bracket depth limit to <i>n</i>	2.0
<code>-build_module_interface_unit</code>	build a C++ module interface unit	2.1
<code>-cc_limit</code>	set the constexpr call depth limit to <i>n</i>	2.0
<code>-cond</code>	conditionally execute options	1.0
<code>+cpp</code>	add C++ extension(s)	—
<code>-cpp</code>	remove C++ extension(s)	—
<code>-cs_limit</code>	set the constexpr step limit to <i>n</i>	2.0
<code>+d</code>	define the preprocessor symbol <i>Name</i> resistant to change via <code>#define</code>	—
<code>++d</code>	define the preprocessor symbol <i>Name</i> that cannot be <code>#undef</code> 'd	—
<code>-d</code>	define the preprocessor symbol <i>Name</i> with value <i>Value</i>	—
<code>-deprecate</code>	deprecates the use of <i>Name</i> within <i>Category</i>	—
<code>-dump_message_list</code>	dumps PC-lint Plus message list to the provided file	1.0
<code>-dump_messages</code>	dumps PC-lint Plus messages to the provided file in the specified format	1.0
<code>+dump_queries</code>	enable dumping of parsed Query ASTs	2.0
<code>-dump_queries</code>	disable dumping of parsed Query ASTs	2.0
<code>!e</code>	disables message <code>#</code> for the current line	—
<code>-e(</code>	inhibits message <code>#s</code> for the next expression	—
<code>--e(</code>	inhibits message <code>#s</code> for the entire enclosing expression	—
<code>+e</code>	re-enables message(s) <code>#</code>	—
<code>-e</code>	disables a message where <code>#</code> is a message number or pattern	—
<code>-e{</code>	inhibits message <code>#s</code> for the next statement	—
<code>--e{</code>	inhibits message <code>#s</code> for the entire enclosing braced region	—
<code>+ecall</code>	enables the message <code>#s</code> within calls to <i>Function</i>	—
<code>-ecall</code>	inhibits the message <code>#s</code> within calls to <i>Function</i>	—
<code>+efile</code>	enables the message <code>#s</code> within <i>File</i>	—
<code>-efile</code>	inhibits the message <code>#s</code> within <i>File</i>	—
<code>+efreeze</code>	freeze the message <code>#s</code> and/or warning level(s)	—
<code>++efreeze</code>	deep freeze the message <code>#s</code> and/or warning level(s)	—
<code>-efreeze</code>	unfreeze the message <code>#s</code> and/or warning level(s)	—
<code>+efunc</code>	enables the message <code>#s</code> within the body of <i>Function</i>	—
<code>-efunc</code>	inhibits the message <code>#s</code> within the body of <i>Function</i>	—
<code>+egrep</code>	enables the message <code>#s</code> when the message text matches <i>Regex</i>	1.0

Option	Summary	Version
<code>-egrep</code>	inhibits the message <i>#s</i> when the message text matches <i>Regex</i>	1.0
<code>+elib</code>	enables message <i>#s</i> in library code	—
<code>-elib</code>	disables the message <i>#s</i> in library code	—
<code>+elibcall</code>	enables message <i>#s</i> inside calls to library functions	—
<code>-elibcall</code>	inhibits message <i>#s</i> inside calls to library functions	—
<code>+elibmacro</code>	enables message <i>#s</i> issued for library macros	—
<code>-elibmacro</code>	inhibits message <i>#s</i> issued for library macros	—
<code>+elibsym</code>	enables message <i>#s</i> issued for library symbols	—
<code>-elibsym</code>	inhibits message <i>#s</i> issued for library symbols	—
<code>+emacro</code>	enables message <i>#s</i> within macro expansions	—
<code>-emacro</code>	inhibits message <i>#s</i> within macro expansions	—
<code>--emacro</code>	inhibits message <i>#s</i> within macro expansions	—
<code>-env_pop</code>	pop the current option environment	1.0
<code>-env_push</code>	push the current option environment	1.0
<code>-env_restore</code>	restore the option environment to a previously saved one	1.0
<code>-env_save</code>	save the current option environment with name <i>Name</i>	1.0
<code>+equery</code>	enables the message <i>#s</i> when <i>Query</i> matches	2.0
<code>-equery</code>	inhibits the message <i>#s</i> when <i>Query</i> matches	2.0
<code>+estring</code>	enables the message <i>#s</i> parameterized by <i>String</i>	—
<code>-estring</code>	inhibits the message <i>#s</i> parameterized by <i>String</i>	—
<code>+esym</code>	enables the message <i>#s</i> parameterized by <i>Symbol</i>	—
<code>-esym</code>	inhibits the message <i>#s</i> parameterized by <i>Symbol</i>	—
<code>+etype</code>	enables the message <i>#s</i> parameterized by <i>Type</i>	—
<code>-etype</code>	inhibits the message <i>#s</i> parameterized by <i>Type</i>	—
<code>-exitcode</code>	set the exit code to <i>n</i>	1.0
<code>+ext</code>	set the extensions to try for extensionless files	—
<code>+f</code>	turns a flag on	—
<code>++f</code>	increments a flag	—
<code>-f</code>	turns a flag off	—
<code>--f</code>	decrements a flag	—
<code>-fallthrough</code>	ignores switch case fallthrough when used in a lint comment	—
<code>+fatal_error</code>	triggers an unsuppressible fatal error using message 399	1.3
<code>-fatal_error</code>	triggers a suppressible fatal error using message 398	1.3
<code>-father</code>	a stricter version of <code>-parent</code>	—
<code>-format</code>	sets the message format for height 3 or less	—
<code>-format4a</code>	sets the format of the message that appears above the error for height 4	—
<code>-format4b</code>	sets the format of the message that appears below the error for height 4	—
<code>-format_category</code>	set format for message category	1.2
<code>-format_intro</code>	sets the format of the line that appears before each new message set	—
<code>-format_stack</code>	sets the format of the stack usage message	—
<code>-format_summary</code>	format of the output produced by the <code>-summary</code> option	—
<code>-format_verbosity</code>	sets the format of verbosity output	—
<code>-function</code>	copy or remove semantics from <i>Function0</i>	—
<code>+group</code>	adds messages from <i>Pattern</i> to message group <i>Name</i>	1.0
<code>-group</code>	remove <i>Pattern</i> from group <i>Name</i> or delete <i>Name</i>	1.0

Option	Summary	Version
<code>-h</code>	adjusts message height options	—
<code>-header</code>	auto-include <i>Filename</i> at the beginning of each module	—
<code>--header</code>	clears previous auto-includes and optionally adds a new one	—
<code>+headerwarn</code>	causes message #829 to be issued when <i>Filename</i> is <code>#included</code>	—
<code>-help</code>	display detailed help about <i>Option</i>	1.0
<code>+html</code>	emit HTML output	—
<code>-i</code>	add search <i>directory</i> for <code>#include</code> directives	—
<code>--i</code>	add lower-priority search <i>directory</i> for <code>#include</code> directives	—
<code>-ident</code>	add identifier characters	—
<code>-idlen</code>	specifies the number of meaningful characters in identifier names	—
<code>-incvar</code>	change the name of the INCLUDE environment variable to <i>Name</i>	—
<code>-index</code>	establish <i>ixtype</i> as index type	—
<code>-indirect</code>	process <i>File</i> as an options file	—
<code>-lang_limit</code>	specify minimum language translation limits	1.0
<code>+libclass</code>	add class of headers treated as libraries	—
<code>+libdir</code>	specify a <i>Directory</i> of headers to treat as libraries	—
<code>-libdir</code>	specify a <i>Directory</i> of headers to not treat as libraries	—
<code>+libh</code>	specify <i>Headers</i> to treat as libraries	—
<code>-libh</code>	specify <i>Headers</i> to not treat as libraries	—
<code>+libm</code>	specify <i>Modules</i> to treat as libraries	—
<code>-libm</code>	specify <i>Modules</i> to not treat as libraries	—
<code>-library</code>	indicates the next source module is to be treated as library code	—
<code>++limit</code>	locks in the message limit at <i>n</i>	—
<code>-limit</code>	set the message limit to <i>n</i>	—
<code>+lnt</code>	add indirect file extension(s)	—
<code>-lnt</code>	remove indirect file extension(s)	—
<code>-locker_tag</code>	specify alternate locker tag class names	1.4
<code>-max_threads</code>	set the maximum number of concurrent threads for parallel analysis	1.0
<code>+message</code>	emits a custom message with the specified message #	—
<code>-message</code>	emits a custom message via info 865	—
<code>+metric</code>	create, check, or nominate a metric	2.0
<code>+metric_report</code>	enable metric report	2.0
<code>+misra_interpret</code>	enable MISRA interpretation	1.2
<code>-misra_interpret</code>	disable MISRA interpretation	1.2
<code>-mutex_attr</code>	specify shared/recursive values used for pthread_mutexattr_settype	1.4
<code>-mutex_init</code>	specify alternate pthread mutex initialization macro names	1.4
<code>+oe</code>	redirect stderr to <i>Filename</i> in append mode	—
<code>-oe</code>	redirect stderr to <i>Filename</i> overwriting existing content	—
<code>+os</code>	redirect stdout to <i>Filename</i> in append mode	—
<code>-os</code>	redirect stdout to <i>Filename</i> overwriting existing content	—
<code>-p</code>	just preprocess	—
<code>+paraminfo</code>	include parameter information for specified messages as verbosity	1.0
<code>-paraminfo</code>	exclude parameter information for specified messages as verbosity	1.0
<code>-parent</code>	augment strong type hierarchy	—
<code>+pch</code>	designates a given header as the pre-compiled header, forcing recreation	—



Option	Summary	Version
-pch	designates a given header as the pre-compiled header, creating precompiled form if needed	—
-pp_sizeof	set the value that <code>sizeof(<i>Text</i>)</code> evaluates to in a preprocessor directive	1.0
+ppw	enable preprocessor keyword(s)	—
-ppw	disable preprocessor keyword(s)	—
--ppw	remove built in meaning of preprocessor keyword(s)	—
-ppw_asgn	assign preprocessor word meaning of <i>Word2</i> to <i>Word1</i>	—
+pragma	associates <i>Action</i> with <i>Identifier</i> for <code>#pragma</code>	—
-pragma	disables pragma <i>Identifier</i>	—
-printf	specified <i>names</i> are <code>printf</code> -like functions with format provided in the <i>N</i> th argument	—
-restore	restores the state of error inhibition settings	—
+rw	enable reserved word(s)	—
-rw	disable reserved word(s)	—
--rw	remove built in meaning of reserved word(s)	—
-rw_asgn	assigns reserved word meaning of <i>Word2</i> to <i>Word1</i>	—
-s	set the size of various types	—
-save	saves the current state of error inhibition settings	—
-scanf	specified <i>names</i> are <code>scanf</code> -like functions with format provided in the <i>N</i> th argument	—
-sem	associates the semantic <i>Sem</i> with <i>Function</i>	—
-setenv	set environment variable <i>name</i> to <i>value</i>	—
-size	set static or auto size thresholds	—
-skip_function	skips the body of a <i>Function</i> when parsing	1.0
-specific_climit	maximum number of specific calls per function	—
+stack	enable stack reporting	—
-stack	set stack reporting options	—
-std	specifies the C or C++ language version	1.0
-strong	imbues typedefs with strong type checking characteristics	—
-subfile	process just options or just modules from options file <i>File</i>	—
-summary	outputs a message summary at the end of processing, optionally to a file	—
-t	sets tab width	—
-thread_report	enable a thread analysis report	1.4
-tr_limit	set the template recursion limit to <i>n</i>	—
+typename	includes the types of symbols when emitting specified messages	—
-typename	excludes the types of symbols when emitting specified messages	—
-u	undefine the preprocessor symbol <i>Name</i>	—
--u	ignore past and future <code>#defines</code> of the preprocessor symbol <i>Name</i>	—
-unit_check	unit checkout	1.0
--unit_check	unit checkout and ignore modules in lower .lnt files	1.0
-unreachable	ignores unreachable code when used in a lint comment	—
+v	output verbosity to stderr and stdout	—
-v	turn off verbosity or send it to stdout	—
-verbofify	print <i>string</i> as a verbosity message	1.0
-vt_depth	specifies the maximum number of nested specific walks	1.0
-vt_passes	specifies the number of passes for intermodule value tracking	1.0
-w	sets the base warning level	—

Option	Summary	Version
<code>-width</code>	sets the maximum output width and indentation level for continuations	—
<code>-wlib</code>	sets the base warning level for library code	—
<code>-write_file</code>	write <i>String</i> to file <i>Filename</i>	1.0
<code>+xml</code>	activates XML escape sequences	—
<code>-zero</code>	sets exit code to 0	—
<code>+zero_err</code>	specify message numbers that should not increment exit code	—
<code>-zero_err</code>	specify message numbers that should increment exit code	—

## 4.3 Message Options

### 4.3.1 Error Inhibition

`-e#` disables a message where `#` is a message number or pattern

`+e#` re-enables message(s) `#`

For example, `-e504` will turn off error message 504. The number designator may contain the wild card characters `?` (single character match) or `*` (multiple character match). For example `-e7??` will turn off all 700 level errors.<sup>3</sup>

As another example:

```
-e1*
```

suppresses all messages beginning with digit 1. This includes messages 12, 1413 and 1 itself. The use of wild card characters is also allowed in `-esym`, `-elib`, `-elibsym`, `-efile`, `-efunc`, `-emacro`, `-e(#)`, `--e(#)`, `-e{#}` and `--e{#}`.

`!e#` disables message `#` for the current line

One-line message suppression (where `#` is a message number) is designed to be used in a `/*lint` or `//lint` comment. It serves to suppress the given message for one line only. For example:

```
if( x = f(34) ) //lint !e720
    y = y / x;
```

will inhibit message 720 for one line. This takes the place of having to use two separate lint comments as in:

```
//lint -save -e720
if( x = f(34) )      //lint -restore
    y = y / x;
```

Multiple error message suppression options are permitted as in the following, but not wild card characters.

```
n = u / -1; //lint !e573 !e713
```

A limitation is that the one-line message suppression may not be placed within macros. This is done for speed. A rapid scan is made of each non-preprocessor input line to look for the character `'!'`. If this

<sup>3</sup>To turn off Informational messages it is better to use `-w2`.

option could be embedded in a macro, such a rapid search could not be done.

The supplemental messages 893, 894, and 897 are immune to direct one-line suppressions. A message suppressed by a one-line suppression, as with any other form of suppression, will also suppress any supplemental messages that would have accompanied it (including those immune to direct one-line suppression).

`-e(# [,#...])` inhibits message #s for the next expression

This is presumably used within a lint comment. For example:

```
a = /*lint -e(413) */ *(char *)0;
```

will inhibit Warning 413 concerning the use of the Null pointer as an argument to unary \*. Note that this message inhibition is self-restoring so that at the end of the expression, Warning 413 is fully restored. Because of this restoration, there is no need for an option `+e(#)`.

This method of inhibiting messages is to be preferred over the apparently equivalent:

```
a = /*lint -save -e413 */ *(char *)0
    /*lint -restore */;
```

The phrase 'next expression' may require further elaboration. It may suffice to say that there should be no surprises. In particular, it may be any fully parenthesized expression, any function call, array reference, structure reference, or unary operators applied to such expressions. It will stop short of any unparenthesized binary (or ternary) operator.

The `-e(`, `--e(`, `-e{`, and `--e{` options are only effective for messages issued during the analysis phase, they cannot be used to suppress messages issued during the preprocessing phase or the semantic analysis phases. This is because expressions do not exist during the preprocessing phase and expression and statement ranges are not established until after semantic analysis has completed. If these options are used with an inappropriate message number, a bad option error will be emitted.

`--e(# [,#...])` inhibits message #s for the entire enclosing expression

For example:

```
a = /*lint --e(413) */ *(int *)0 + *(char *)0;
```

will inhibit both Warning 413's that would normally occur in the given expression statement. Had the option `-e(413)` been used instead of `--e(413)` then only the first Warning 413 would have been inhibited.

The *entire expression* can be an `if` clause, a `while` clause, any one of the `for` clauses, a `switch` clause or an expression statement.

The limitations described for the `-e(#)` option above apply to this option as well.

`-e{# [,#...]}` inhibits message #s for the next statement

This is presumably used within a lint comment. Consider the following example:

```
//lint -e{715} suppress "k not referenced"

void f(int n, unsigned u, int k)    // 715 not issued
{
    //lint -e{732} suppress "loss of sign"
    u = n;                          // 732 not issued
}
```

```

        //lint -e{713} suppress "loss of precision"
        if (n) {
            n = u;    // 713 not issued
        }
    }
}

```

The `-e{715}` is used to suppress message 715 over the entire function but not subsequent functions. The `-e{732}` is used to suppress message 732 in the assignment that follows. The `-e{713}` is used to suppress message 713 over the entire `if` statement that follows.

Note that this construct can be used before a class definition or namespace declaration to inhibit messages associated with that class or namespace body.

Wild card characters may be used with `-e{}`. Thus `-e{7??}` or `-e{*}` are legitimate.

Note that `-emacro( {#}, symbol )` will indirectly result in the `-e{#}` being used. See `-emacro({#}, symbol)`

Note that since `/*lint */` options that appear in macros are retained you may define a macro for error suppression that is parameterized by number. Given the definition:

```
#define Suppress(n) /*lint -e{n} */
```

then,

```
    Suppress(715)
```

will suppress message 715 for the next statement or declaration.

The limitations described for the `-e(#)` option above apply to this option as well.

`--e{# [,#...]}` inhibits message `#s` for the entire enclosing braced region

This option must be used within a lint comment. The supplied message pattern will be suppressed within the entirety of the nearest enclosing braced region where the comment is placed. A braced region may be a compound statement, function, class, struct, union, or namespace. If the option is not placed within any such braced region, the suppression applies to the module as a whole. Like other options found in lint comments, it does not extend past the end of the module into the next module.

The limitations described for the `-e(#)` option above apply to this option as well.

`-ecall(# [#...], Function [,Function...])` inhibits the message `#s` within calls to *Function*  
`+ecall(# [#...], Function [,Function...])` enables the message `#s` within calls to *Function*

This includes the parsing of the function call and any of the arguments to the call. Example:

```

//lint -ecall(713,f)
void f(int);
void h(int);
void g(unsigned u) {
    h(u);    // elicits 713: "Loss of precision"
    f(u);    // 713 suppressed.
}

```

Please note the distinction between `-ecall` and `-efunc`. The former suppresses within a call expression whereas the latter suppresses within the definition.

`-efile(# [#...], File [,File...])` inhibits the message *#s* within *File*  
`+efile(# [#...], File [,File...])` enables the message *#s* within *File*

This option is used to suppress messages from being issued within specific files.

For example,

```
-efile(714, core.c)
```

will suppress message 714 for symbols defined in `core.c`

The name provided must match the file name as reported by PC-lint Plus. In particular, if the `ffn` flag is ON, the full file name is expected to be provided. To explicitly match the base name of the file (with all directory information removed), prefix the filename with a minus (-). Similarly, to explicitly match the full path of the file (as would be reported with `+ffn`), prefix the filename with a plus (+). For example:

```
-efile(714, core.c)
```

will not work if using `+ffn` and will not suppress 714 if the file is reported by PC-lint Plus with a directory such as `foo/core.c`. The option: `-efile(714, -core.c)` will suppress 714 within any file whose name is `core.c`, regardless of its location or the value of `ffn`. Finally:

```
-efile(714, +/a/b/core.c)
```

will suppress 714 only within `core.c` located in `/a/b/`. A file whose name begins with - or + can be suppressed by prefixing the name with a backtick(`), e.g. `-efile(714, `+file.c)`. Wildcards may be used in the File pattern.

`+efreeze | +efreeze(#|w# [,#|w#...])` freeze the message *#s* and/or warning level(s)  
`-efreeze | -efreeze(#|w# [,#|w#...])` unfreeze the message *#s* and/or warning level(s)  
`++efreeze | ++efreeze(#|w# [,#|w#...])` deep freeze the message *#s* and/or warning level(s)

It is sometimes useful to inhibit error suppression options so that the programmer can view what messages had been suppressed. The `+efreeze` and `++efreeze` options are designed to do precisely this. These options place some or all messages in a *frozen state*. Frozen messages cannot be the target of *subsequent* suppression options, i.e. the following options are ignored when applied to a frozen message:

```
-e
!e
-ecall
-efile
-efunc
-egrep
-elib
-elibcall
-elibmacro
-elibsym
-emacs
-estring
-esym
-etyp
```

Additionally, moving to a lower warning level with the `-w` and `-wlib` options will not suppress frozen messages. Freezing a message doesn't implicitly enable the message and doesn't prevent the message from being enabled with e.g. `+e#`, it only inhibits suppression options encountered while the message is frozen. Note also that only suppression options encountered while the message is frozen are affected, e.g.

a `-esym` option can affect the issuance of a message that is not frozen until after the `-esym` is encountered.

For example, let file `x.c` contain:

```
int f( int n )
{ return n & 0; } //lint !e835 suppress Info 835
```

Normal linting of `x.c` will not show the ending of a 0 because message 835 has been suppressed with the `!e835`. However if our command line consists of:

```
lint +efreeze x.c
```

the suppression itself is suppressed and the message will be issued.

The programmer can emerge (presumably temporarily) from a frozen state by using the option `-efreeze`.

```
//lint -save -efreeze
#include <lib.h>
//lint -restore
```

In this example, we will temporarily emerge from the frozen condition for the duration of processing `lib.h`. For this to work the freeze status is one of the settings affected by the `-save` options.

A super cooled state can be created with the `++efreeze` option. This will not admit to any attempt at a thaw. If `++efreeze` had been used prior to the above example, the attempt to use `-efreeze` would have had no effect.

Without any arguments, these options apply to all messages. Their effects can be limited to individual message numbers or messages within a particular warning level by supplying these as arguments. For example, `++freeze(534)` will prevent later-appearing options from suppressing message 534 without affecting the frozen status of other messages. To inhibit suppression of error messages, `++efreeze(w1)` can be used.

`-efunc(# [#...], Function [,Function...])` inhibits the message #s within the body of *Function*

`+efunc(# [#...], Function [,Function...])` enables the message #s within the body of *Function*

For example:

```
int f(int n)
{
    return n / 0;    // Error 54
}
```

will result in message 54 (divide by 0). To inhibit this message you may use the option:

```
-efunc( 54, f )
```

This will, of course, inhibit any other divide by 0 message occurring **within** the same function.

The `-efunc` option contrasts with the `-esym` option, which suppresses messages **about** a named function or, indeed, **about** any named symbol, which is parameterized by *Symbol* within the error message.

Both the error number and the *Function* may contain wild card characters. See the discussion of the `-esym` option.

Member functions must be denoted using the scope operator. Thus:

```
-efunc( 54, X::operator= )
```

inhibits message 54 within the assignment operator for class X but not for any other class. Creative uses of the wild card characters can be employed to make one option serve to suppress messages over a wide range of functions, such as all assignment operators, or all member functions within a class. See the discussion in [-esym](#).

There are times when you might want to quote the 2nd argument to `efunc` and/or escape some of the pattern valued characters. See [-esym...](#) for an explanation and examples.

`-egrep( # [#...], Regex [,Regex...]` inhibits the message #s when the message text matches *Regex*  
`+egrep( # [#...], Regex [,Regex...]` enables the message #s when the message text matches *Regex*

The `-egrep` option will suppress messages when the supplied regular expression matches the text of the message. This is particularly useful when it is desired to suppress a message based on the values of multiple parameters.

Below are some points to consider when employing `-egrep`:

- The `+egrep` option works the same way as `-egrep` but is used to enable messages.
- `-egrep` uses regular expressions, not wild cards, to perform the match. In particular,
  - the wildcard `*` regular expression equivalent is `.*`
  - the wildcard equivalent of `?` is `.?`
  - the wildcard equivalent of `[abc]` is `(abc)?`
- Whereas the parameterized suppression options match the full text of the parameter, the `-egrep` option by default matches any part of the message. An anchored match can be accomplished using the `^` and `$` anchoring characters at the start and end of the regular expression.
- The text that is matched is the text that corresponds to the `%m` specifier in the `-format` option. This includes the full message text (after parameter substitution) as well as any appended text introduced via the `-append` option but not text injected via the `+typename` option.
- The PCRE (Perl) regular expression syntax is used for the regular expressions supported by `-egrep`.

For example, message 9078 (cast between `pointer type Type` and `integer type Type`) is given whenever there is a cast between `pointer` and `integer` types. If it is desired to suppress the message only for converting to an `int`, there is no way to accomplish this using `-etype`. `-etype(9078, int)` will suppress the message in such cases but will also suppress cases where an `int` type is converted *from* because there is not a way to specify which of the type parameters the `-etype` option should operate on. In such a case, `-egrep` may be used as in

```
-egrep(9078, "type .* and integer type 'int'$")
```

`-elib( # [,#...] )` disables the message #s in library code

`+elib( # [,#...] )` enables message #s in library code

See Chapter 5 [Libraries](#). This is handy because library headers are usually "beyond the control" of the individual programmer. For example, if the `stdio.h` you are using has the construct

```
#endif comment
```

instead of

```
#endif /*comment*/
```

as it should, you will receive message 544. This can be inhibited for just library headers by `-elib(544)`. `#` may contain wild cards. However, a more convenient way to set a level of checking within library code is via `-wlib(level)`. See also `-elibsym()`.

`-elibcall(# [,#...])` inhibits message `#s` inside calls to library functions  
`+elibcall(# [,#...])` enables message `#s` inside calls to library functions

This option is similar to `-ecall` but suppresses the specified messages from calls to all library functions.

`-elibmacro(# [,#...])` inhibits message `#s` issued for library macros  
`+elibmacro(# [,#...])` enables message `#s` issued for library macros

This option is like `-emacro(#,symbol)` except that it applies to all macros defined in library code. Like `-emacro`, you may use a form beginning with `--`. The `#` may be surrounded with parens or with curly braces and these have the same meaning as with `-emacro`. E.g.

```
//lint -elibmacro({414},{54},835)
//lint ++flb           // Enter Library region
#define A() ( 1 / 0 ) // macro becomes a library macro
//lint --flb           // Leave Library region
int j = A() ;           // No message issued.
```

Please note that:

```
-e123 +elibmacro(123)
```

does not do what you might think. A `+elibmacro` only undoes a previous `-elibmacro` that affected the same message number. There is no way currently of enabling a message for only library macros.

`-elibsym(# [,#...])` inhibits message `#s` issued for library symbols  
`+elibsym(# [,#...])` enables message `#s` issued for library symbols

For example, suppose a library defines a pair of classes:

```
class X { };
class Y : public X { ~Y(); };
```

This will result in message 1790, public base 'X' of 'Y' has no non-destructor virtual functions. Note that the message is deferred until derived class Y is seen. The option `-elib(1790)` will suppress this message while processing library headers. If in the user's own code there is a declaration:

```
class Z : public X { ~Z(); };
```

the diagnostic will be issued in spite of the fact that `-elib(1790)` is given because we are outside library code. The user may suppress this by using `-esym( 1790, X )`. But if there are a large number of such base classes, the user may prefer to issue the option:

```
-elibsym( 1790 )
```

which in effect does an `-esym(1790,s)` for all library header symbols `s`.

Please note that:

```
-e123 +elibsym(123)
```

does not do what you might think. A `+elibsym` only undoes a previous `-elibsym` that affected the same message number. There is no way currently of enabling a message for only library symbols.



`-emacro( # [#...], Symbol [,Symbol...])` inhibits message `#s` within macro expansions  
`+emacro( # [#...], Symbol [,Symbol...])` enables message `#s` within macro expansions  
`--emacro( # [#...], Symbol [,Symbol...])` inhibits message `#s` within macro expansions

Suppresses message `#` in the expansion of the specified macros. The option must precede the macro definition. The option `-emacro( #, symbol, ...)` is designed to suppress message number `#` for each of the listed macros. For example,

```
-emacro( 778, TROUBLE )
```

will suppress message 778 when expanding macro TROUBLE.

Note that the `-emacro` options only suppress messages within macro expansions. In particular, to suppress a message that *mentions* the name of a macro, use `-estring` instead.

There are times when you might want to quote the 2nd argument to `emacro` and/or escape some of the pattern valued characters. See option `-esym ...` for an explanation and examples.

- `-emacro( (#), symbol, ... )` inhibits, for a macro expression,  
`--emacro( (#), symbol, ... )` inhibits, for the entire expression,

Suppresses message `#` in the expansion of the specified macros. The macros are expected to be expressions (syntactically).

The `--` form of this option uses the `--e( #)` option and this inhibits messages in the entire expression in which it is embedded. Thus, the option

```
--emacro( (413), ZERO )
```

would be as if we had defined ZERO as:

```
#define ZERO /*lint --e(413) */ (* (int *) 0)
```

This has the effect of inhibiting this message for the entire expression in which ZERO is embedded.

- `-emacro( {#}, symbol, ... )` inhibits for a macro statement,  
`--emacro( {#}, symbol, ... )` inhibits for a macro within a region,

Suppresses message `#` in the expansion of the specified macros. For example, the `Swap` macro below will swap the values of two integers.

```
//lint -emacro( {717}, Swap )
#define Swap( n, m ) do {int _k = n; n=m; m=_k; } while(0)
```

By using the `do { } while(0)` trick the macro can be employed exactly as any function. However, the trick will engender message 717 because of the '0' in the while. The message can be suppressed using the curly bracket version of the `-emacro` option as shown. This rendition will prefix the body of the macro with the lint comment

```
/*lint -e{717} */
```

Use of the `--emacro( {#} ...)` option will cause the lint comment

```
/*lint --e{#} */
```

to be prepended to the macro.

`-eqquery(# [#...], Query)` inhibits the message *#s* when *Query* matches  
`+eqquery(# [#...], Query)` enables the message *#s* when *Query* matches

These options are used to control message suppression based on the result of a *Query* that is evaluated at message issuance time. The `-eqquery` option will suppress messages when the supplied *Query* evaluates to a *truthy* value. For example:

```
-eqquery(916, msgTypeParam(2).isVoidPointerType())
```

will suppress message [916](#) when the message would be issued with a void pointer as the second type parameter (the description of each message includes the available parameterizations). If the message is anchored to a statement or expression, the corresponding AST node may be inspected within the query. For example, message [523](#) is issued at a location anchored to the expression that is determined not to have side effects. The following option will suppress this message when it is issued at the site of a call to a virtual function:

```
-eqquery(523, CallExpr().getDirectCallee().CXXMethodDecl().isVirtual())
```

The `+paraminfo` option may be used to determine whether a particular message is issued with an accessible AST node.

`-estring(# [#...], String [,String...])` inhibits the message *#s* parameterized by *String*  
`+estring(# [#...], String [,String...])` enables the message *#s* parameterized by *String*

Consider the following example:

```
int f(unsigned char c) {
    if (c < 1000) return 1;
    else return 0;
}
```

This will result in the following message:

```
warning 650: constant '1000' out of range for operator '<'
    if (c < 1000) return 1;
        ^
```

If we examine message [650](#), we see that it is parameterized by an *integer* and a *string*. We may use `-estring` to suppress on the basis of either of these parameters, e.g.:

```
-estring(650, "<")
```

The second argument to `-estring` means that the 650 will not be issued when the operation (represented by the string parameter) is `<`.

The `-estring` option may be used with messages that are parameterized by anything other than *type* or *symbol*.

`-esym(# [#...], Symbol [,Symbol...])` inhibits the message *#s* parameterized by *Symbol*  
`+esym(# [#...], Symbol [,Symbol...])` enables the message *#s* parameterized by *Symbol*

This is one of the more useful options because it inhibits messages with laser-like precision. For example `-esym(714,alpha,beta)` will inhibit error message [714](#) for symbols `alpha` and `beta`. Messages that are parameterized by the identifier *Symbol* can be so suppressed. Also, if the `fsn` flag is ON (See Section [4.11 Flag Options](#)) messages parameterized by *String* may also be suppressed. Thus, if you examine Message [714](#) you will notice that the italicized word '*Symbol*' appears in the text of the message.

It is possible to macroize a lint option in order to remove some of the ugliness. The following macro `NOT_REF(x)` can be used to suppress message [714](#) about any variable `x`.

```
#define NOT_REF(x)    /*lint -esym( 714, x ) */
...
NOT_REF( alpha )
int alpha;
```

For C++, when the *Symbol* appearing within a message designates a function, the function's signature is used. The signature consists of the fully qualified name followed, by a list of parameter types (i.e. the full prototype). For example, in the unlikely case that a C++ module contained only:

```
class X    {
    void f(double, int); { }
};
```

the resulting messages would include:

```
info 1714: member function 'X::f(double, int)' not referenced
```

To suppress this message with `-esym` you must use:

```
-esym( 1714, X::f )
```

The full signature of the Symbol is `X::f(double, int)`. However, its name is `X::f`, and it is the name of the symbol (signature minus any arguments) that is used for error suppression purposes. Please note that the unqualified name may not be used. You may not, for example, use

```
-esym( 1714, f )
```

to suppress errors about `X::f`.

A `+esym` can be used to override a `-e#` just as a `-esym` can override a `+e#`. Thus, an option combination like:

```
-e714 +esym( 714,alpha )
```

will cause **714** to be reported only for `alpha`.

The suppression (or the enabling) of `esym` is weighted against the options: `-e#`, `+e#`, `-elibsymb`, `-efunc`, `-etype`, `-estring`, `-ecall` and `+efunc`. When Lint is about to report a message, it tallies the "votes" from these options (inasmuch as they apply to the current message). Each applicable option beginning with a '-' counts as a vote of -1; each beginning with a '+' counts as +1. Since several symbols and names can parameterize a message, it is necessary to tally the negative and positive contributions of all appropriate `-esym` and `+esym` options. If the net result is less than zero, the message is suppressed. For example:

```
-esym(648,a*) +esym(648,apple)
```

will suppress **648** for all symbols beginning with 'a' except for "apple".

There are times when you might want to quote the 2nd argument to `esym` and/or escape some of the pattern valued characters. See Section [4.1.6 Quotes in Arguments](#) for an explanation and examples.

`-etype(# [#...], Type [,Type...])` inhibits the message `#s` parameterized by *Type*  
`+etype(# [#...], Type [,Type...])` enables the message `#s` parameterized by *Type*

Both `#` and the *Type* parameters may contain wild card characters. This option is similar to `-esym` except that it operates on the name of the symbol's type as opposed to the name of the symbol. It must be emphasized that this option applies only to *Symbol* and *Type* parameters, not *Name* parameters or other kinds of parameters.

The representation that PC-lint Plus uses to denote a symbol's type can be obtained by using `+typename(#)` where `#` is a message number (or pattern). Note, that it is not necessary to use `+typename` to inhibit messages with `-etype`. Example:

```
//lint -etype(1746, FooSmartPtr<*>)
template <class T> class FooSmartPtr {};
void f(FooSmartPtr<int> a) {}
// 1746 ("Parameter 'a' could be made const reference") suppressed.
```

Note that it is possible to suppress a message globally and then enable it for a specific type or types by using the `+etype` form of the option. See the description of `+esym`. This, of course, presumes that the message has a symbol or type parameter.

`+group(Name [,Pattern...])` adds messages from *Pattern* to message group *Name*  
`-group(Name [,Pattern...])` remove *Pattern* from group *Name* or delete *Name*

A group of messages can be given a name that can be used anywhere that a message number pattern is allowed. The `+group` option is used to create a new named group or add messages to an existing group. For example,

```
+group(formats, 495, 496, 497)
```

will create a message group named `formats` that contains the messages `495`, `496`, and `497`. Messages can be added to this group with additional `+group` options, e.g.

```
+group(formats, 240?)
```

will add the messages 2400–2409 to the `formats` group. The `-group` option can be used to remove messages from a group, for example,

```
-group(formats, 2400)
```

will remove `2400` from the `formats` group.

Group names can be used in an option where a message number pattern is accepted. For example

```
-esym(formats, vsprintf), -eformats
```

`-group(name)` without any message patterns will delete the named group. For example:

```
-group(formats)
```

will remove the group named `formats`. This is different from removing all messages in a group, which leaves an empty group that may still be referenced in other options. Referencing a deleted group will result in an error.

Group names may contain upper and lower case letters, digits, `.`, `-`, and `_` but must start with a letter. Group names are considered to be global and are not part of the Option Environment.

`-restore` restores the state of error inhibition settings

This option restores the state of the error inhibitions settings (see `-save`) to their state at the start of the last `-save`. For example:

```
/*lint -save -e641 */
    some code
/*lint -restore */
```

temporarily suppresses Warning `641`. It is better to restore `641` this way than with a `+e641` because if `641` had been turned off globally (say, at the command line) this sequence would not accidentally turn it back on. `-restore` will also pop the most recent `-save` if any, so that `-save -restore` sequences can be nested. If there is no corresponding `-save` in effect at the time `-restore` is used, an error will be issued.

`-save` saves the current state of error inhibition settings

The error inhibition settings affected consist of those set with the following options:

```
-e#
+e#
+efreeze
-efreeze
-w#
```

`-save` can be used in a recursive option inhibition setting. For example,

```
#define alpha \
/*lint -save -e621 */ \
something \
/*lint -restore */
```

within macro `alpha` will suppress message 621 setting without affecting either the error suppression state or other `-save`, `-restore` options. There is no intrinsic limit to the number of successive `-save` options.

A `-save` can be used on the command line or in a `.lnt` file. E.g., suppose we have two modules, `divzero1.c` and `divzero2.c`, and suppose both modules contain the expression `(1/0)`, which normally elicits both error 54 ("Division by zero") and warning 414 ("Possible division by zero"). Then, if our project's `.lnt` file contains:

```
-e414
-save
-e54
divzero1.c
-restore
divzero2.c
```

... then PC-lint Plus will issue neither error 54 nor warning 414 while processing `divzero1.c`. While processing `divzero2.c`, warning 414 will still be suppressed (because of the `-e414` that was issued before the `-save`), but PC-lint Plus will issue error 54 because that message was not suppressed at the time that we issued the `-save` to which the `-restore` option corresponds.

If you have more `-restore` options than `-save` options within a module, then the extra `-restore` options will revert the error state back to what was in effect before the last `-save` encountered outside a module, if any. If there is no corresponding `-save`, either inside or outside the module, when a `-restore` option is encountered, an error will be issued.

Like all options encountered inside a module, the effects of `-save` and `-restore` within a module are limited to the module in which they appear. Once the module is finished processing, the suppression state of the next module is the same as it was before the previous module was analyzed regardless of options encountered inside the previous module.

See also `-env_push` and `-env_pop` which provide similar functionality but affect a larger portion of the option environment.

`-w#` sets the base warning level

The warning levels are:

- `-w0` No messages (except for fatal errors).
- `-w1` Error messages only – no Warnings or Informationals.
- `-w2` Error and Warning messages only.
- `-w3` Error, Warning and Informational messages (this is the default).
- `-w4` All messages (including Elective Notes).

The default warning level is level 3.

The option `-w#` will establish a new warning level and affect only those messages in the *zone of transition* (see below). Thus, the option:

```
-e521 -e50 -w2
```

will have the effect of suppressing 521, 50 and all Informationals. On the other hand

```
-e521 -e50 -w1 -w2
```

will suppress 50 and all Informationals. Warning 521 will be restored by the `-w2` because Warnings are in the zone of transition in going from level 1 to 2.

Because options are processed in order, the combined effect of the two options: `-w2 +e720` is to turn off all Informational messages except 720. Specifying an invalid warning level (one not in the above table) will result in error 72.

### Zones of Transition

A *zone of transition* is the set of messages that are impacted by moving to a different warning level or library warning level with the `-w` and `-wlib` options.

When moving to a higher warning level with `-w` or a higher library warning level with `-wlib`, the zone of transition is the set of messages in warning levels (*prev-level*, *new-level*], the ( representing an exclusive bound and ] representing an inclusive bound. For example, if the current warning level is 1 and the `-w3` option is used, the zone of transition constitutes the messages in warning levels (1, 3] or levels 2 and 3. Moving to a higher warning level *enables* all messages in the zone of transition.

When moving to a lower warning level, the zone of transition is [*4*, *new-level*) for `-w` and [*prev-level*, *new-level*) for `-wlib`. Moving to a lower warning level *suppresses* all messages in the zone of transition (except enabled messages that have been frozen with `+efreeze` or `++efreeze`). For example, after the options `-w3 +e9001 -w2`, all messages in warning levels 4 and 3 will be suppressed, including message 9001 because the zone of transition is [4, 2).

Note that using `-w` or `-wlib` with the current warning level or library warning level results in an empty zone of transition and as a result has no effect and will result in warning 686. To suppress all messages in a warning level higher than the current level, first raise the warning level and then lower it to create a zone of transition. For example, if the current library warning level is 1, the options `-wlib(4) -wlib(1)` will suppress all non-error messages for library regions. This is useful e.g. to suppress MISRA messages from library code when using one of the provided MISRA author configurations.

`-wlib(#)` sets the base warning level for library code

It will not affect C/C++ source modules. The warning *Levels* may have the same range of values as `-w#` and are as follows:

- `-wlib(0)` No library messages
- `-wlib(1)` Error messages only (when processing library code.) This is the default
- `-wlib(2)` Errors and Warnings only
- `-wlib(3)` Error, Warning and Informational.
- `-wlib(4)` All messages (not otherwise inhibited).

For example,

```
-wlib(2)
```

is equivalent to

```
-elib(7??) -elib(8??) -elib(9??)
-elib(17??) -elib(18??) -elib(19??)
-elib(27??) -elib(28??) -elib(29??)
-elib(37??) -elib(38??) -elib(39??)
-elib(8???) -elib(9???)
```

but easier to type.

Many users complain that they do not wish to be informed of 'lint' within library headers. In general, you may use `-elib` to repeatedly inhibit individual messages but this may prove to be a tedious exercise if there are many different kinds of messages to inhibit. Instead you may use

```
-wlib(1)
```

to inhibit all library messages except syntactic errors.

See the section "Zones of Transition" in the description of the `-w` option for details about which messages are affected, and how, when moving between library warning levels.

### 4.3.2 Verbosity

`-v[acehiotuw#] [mf<int>]` turn off verbosity or send it to stdout

`+v[acehiotuw#] [mf<int>]` output verbosity to stderr and stdout

Verbosity refers to the frequency and kind of work-in-progress messages. Verbosity can be controlled by options beginning either with `-v` or with `+v`.

If `-v` is used, the verbosity messages are sent to standard out. On the other hand, if `+v` is used, the verbosity messages are sent to both, standard out and to standard error. This is useful if you are redirecting error messages to a file and want to see verbosity messages at your terminal as well as interspersed with the error messages. For clarity, the options below are given in terms of `-v`.

Except for the option `+v` by itself all verbosity options completely replace prior verbosity settings. It just didn't make sense to treat `+v` as turning off verbosity.

The general format is: `{-+}v[acehiostw#] [mf<int>]`. There may be zero or more characters chosen from the set "acehiostw#". This is followed by exactly one of "mf" or an integer.

- `-vn` (where *n* is some integer) will print a message every *n* lines. This option implies `-vf`. This option will also trigger a file resumption message. Example: `-v1` will issue a message for each line of source code.
- `-va...` Will cause a message to be printed each time there is an Attempt to open a file. This is especially useful to determine the sequence of attempts to open a file using a variety of search directories.
- `-vc...` This will cause a message to be printed each time a function is called with a unique set of arguments. This is referred to as a Specific Call. See Section [8.8 Interfunction Value Tracking](#)
- `-ve...` Will cause a message to be printed each time a template function is instantiated.
- `-vf` Print the names of all source Files as they are encountered. This means all headers as well as module names. Thus, `-vf` implies `-vm`. This option will indicate which headers are "Library Header Files". See Section [5.1 Library Header Files](#).

- vh... At termination of processing the strong type Hierarchy be dumped in an elegant tree diagram. See the example in Section [7.5.1 The Need for a Type Hierarchy](#).
- vh-... The 'h' verbosity flag continues to mean that the strong type hierarchy is dumped upon termination. If the 'h' is followed immediately by a '-' then the hierarchy will be compressed, producing the same tree in half the lines. See Section [7.6 Printing the Hierarchy Tree](#) for an example.
- vi... Output the names of Indirect files (.Int) as they are encountered. This letter 'i' may be combined with others.
- vm Print the names of Modules (referring to C or C++ source files — translation units) as they are encountered, and the names of C++20 module interface units as they are built. (this is the default).
- vo... Output Options as they are encountered whether they are inside lint comments or on the command line. The letter 'o' may be combined with other letters.
- vq... Emit user-defined verbosity messages via the [verbosify](#) operator in [Queries](#).
- vt... The 't' flag may be added to the verbosity option. This will cause a message to be printed each time a template is to be instantiated.
- vu... This verbosity option causes user-defined verbosity messages to be emitted when using the [-verbosify](#) option.
- vw... This verbosity flag will issue a report whenever a function is to be processed with specific arguments. This is called a Specific Walk. See Section [8.8 Interfunction Value Tracking](#).
- v#... The character '#' is usually used with 'f' and will request identification numbers be printed with each file. This is used to determine whether two files are regarded as being the same.

For example:

```
lint +vof file1 file2 >temp
```

will cause a line of information to be generated for each module, each header and each option. This information will appear at the console as well as being redirected into the file `temp`. But not all systems support such redirection. Fortunately, there is the [-os](#) option

```
lint +vof -os(temp) file1 file2
```

[-verbosify\(string\)](#) print *string* as a verbosity message

The [-verbosify](#) option causes the provided string to be emitted as a verbosity message if "user" verbosity output is enabled (which can be accomplished using the verbosity option [-vu](#))

### 4.3.3 Message Presentation

[-append\(#, String\)](#) append *String* to diagnostic # when issued

This option can be used to append a trailing message (*string*) onto an existing error message. For example:

```
-append( 936, - X Corp. Software Policy 371 )
```

will append the indicated message to the text of message [936](#).



The purpose of this option, as the example suggests, is to add additional information, to a message, that could be used to support a company or standards body software policy. Referring to the example above, when message 936 is issued, the programmer can see that this has something to do with Software Policy 371. The programmer can then look up Policy 371 and obtain supplementary information about the practice that is to be avoided.

Note that this option does not automatically enable the indicated message. This would be done separately with, in this example, the option `+e936`. When the form of the option is:

```
-append(errno (name), string)
```

the option is parameterized to append the given text only when certain names appear in the Lint output. For example:

```
-append( 533(elephant), Set this variable to 5 )
```

will append the given text only when message 533 is issued for the preprocessor variable "elephant".

Lastly, multiple `-append()` options will append multiple messages to the specified Lint diagnostic. Consequently:

```
-append( 123, Shop Rule #149 )
-append( 123, Personal Preference #7 )
```

will add "Shop Rule #149, Personal Preference #7" to message 123.

`-format` sets the message format for height 3 or less

`-format4a` sets the format of the message that appears above the error for height 4

`-format4b` sets the format of the message that appears below the error for height 4

This option is especially useful if you are using an editor that expects a particular style of error message. The format option is of the form `-format=...` where the ellipsis consists of any ordinary characters and the following special escapes:

```
%f = the filename (the +ffn flag controls whether full path names are used)
%l = the line number
%t = the message type (Error, Warning, etc.)
%n = the message number
%m = the message text
%c = the column number (bytes from beginning of line)
%C = the column number +1
%% = a percent sign
%(...%) = conditionally include the information denoted by ... if location information is available
\n = newline
\t = tab
\s = space
\a = alarm (becomes ASCII 7)
\" = quote (")
\\ = backslash ( '\ ' )
\T = introduce a real tab into the output
\e = ASCII escape
```

For example the default message format is

```
-"format=%(%f %l %)%t %n: %m"
```

Note that the option is surrounded by quotes so that the embedded spaces do not terminate the option. We could have used `\s` instead, but it is difficult to read.

If the height of the message is 4 (option `-h...4`), the `-format=` option will have no effect. To customize the message use options `-format4a=...` for the line that goes Above the line in error and `-format4b=...` for the line that goes Below.

The `\e` escape sequence can be used to embed ANSI escape codes in the message format to modify output color for terminals that support this. If you intend to include terminal escape sequences in the message format to colorize or otherwise style portions of the output then consider using the `-cond` option with the condition `__is_stdout_terminal` to avoid escape sequences when output is redirected or piped.

Sample configuration for colors in terminals supporting ANSI escape codes:

```
-env_push
+fbe
-format_category(error,"\\e\[1;31m%c")
-format_category(warning,"\\e\[1;33m%c")
-format_category(info,"\\e\[1;34m%c")
-format_category(note,"\\e\[1;35m%c")
-format_category(supplemental,"\\e\[1;37m%c")
-format="%(%f %l %)%t %n\\e\[0m: \\e\[1;1m%m\\e\[0m"
-fbe
-width=120
-env_save(color_format)
-env_pop

-cond(__is_stdout_terminal,
-env_restore(color_format)
)
```

`-format_category(category, string)` set format for message category

This option provides additional control over the expansion of the `%t` (message type) specifier used by the `-format` option. The only format specifier for `-format_category` is `%c` which expands to the name of the relevant message category. The default format is simply `%c`. The result of expanding the format string for this option determines the value `%t` expands to in the format string for `-format`. The first argument is the category whose format string is to be set, one of **error**, **warning**, **info**, **note**, **supplemental**, or **all**. The second argument is the format string.

For example, `pclp -message=hello` will emit:

```
<command line> 2  info 865: hello
-message=hello
^
```

whereas `pclp -format_category[info,abc_%c_xyz] -message=hello` will emit:

```
<command line> 3  abc_info_xyz 865: hello
-message=hello
^
```

`-format_intro` sets the format of the line that appears before each new message set

A message set consists of a primary message and any supplemental messages that are given in association with the message. `-format_intro` can be used to produce a line that appears before every new message set, an empty value (the default) results in no introduction line being printed. For example, to separate

every message set with 4 dashes, you can use `-format_intro=----\n`. The same escape sequences supported by `-format` can be used with `-format_intro`.

`-format_stack` sets the format of the stack usage message

This is the report that deals with stack usage. If this option is not given, a default is assumed.

The option has two uses. It can be used to output information in a form that can readily be absorbed into a database or a spread sheet. It can also be used to obtain a tabular display that is more suitable to visual inspection than the default narrative output.

The format string may contain the following escapes:

`%f` = function name  
`%a` = auto storage requirements  
`%t` = type of function  
`%n` = total stack requirements if computable  
`%c` = function called by `%f`  
`%e` = an indicator as to whether the function called is external  
`%%` = a percent sign

`\n` = newline  
`\t` = tab  
`\s` = space  
`\a` = alarm  
`\q` = quote( " )  
`\\` = backslash

The `%` formats may be immediately followed by a field width (in a manner reminiscent of the `printf` function). If the field width is negative the information is left justified in the field. For example:

```
-"format_stack=%-20f %5a %-20t %5n %c %e"
```

will left-justify the function name and the function type in fields of width 20, and right justify the local stack and total stack requirements in fields of width 5.

`-format_summary` format of the output produced by the `-summary` option

The escape options usable with `-format` are also usable with `-format_summary`.

The available format specifiers are:

`%n` = the message Number  
`%c` = the Count of instances of a message  
`%t` = the message Type  
`%m` = the Message text

The default summary format is:

```
-format_summary="%c\t\t\t%n\t%m"
```

`-format_verbosity` sets the format of verbosity output

Its primary purpose is to allow the user to add font information to the verbosity output. An example of its use can be found in the file `env-html.lnt`.

The format string may contain the following escapes:

`%m` = the normal verbosity message

`\n` = newline

`\t` = tab

`\s` = space

`\a` = alarm

`\q` = quote(" )

`\\` = backslash

`-h[s] [A/B] [a/b] [r] [I] N` adjusts message height options

The optional `s` means Space (blank line) after each message.

The `a` and `b` (meaning respectively Above and Below) refer to the location of the indicator `I` with respect to the source line. This is only used for heights of 3 and 4. The `A` and `B` refer to the position of the context information with respect to the message for message heights 2 and 3, the context appears above the message for `A` and below the message for `B`.

The optional `r` (meaning Repeat) will cause each source line to be repeated for each message produced for the line. This may be preferred for automatic processing of the message file.

The optional `I` stands for a user-designated string of characters to be used as a horizontal position indicator denoting the position of the error within the source line. `I` may not start with `s`, `f`, `r`, `m`, `a` or `b`. This string will be embedded within the source line if `N == 2` (see below) or will appear on its own line if `N > 2`. The indicator may contain digits. This might be useful, for example, in producing an ANSI escape sequence to produce a colored cursor.

The very last digit of the `-h` option is taken to be the height. `N` is an integer in the range 1 to 4 indicating the height of the messages (as further described below). Note that `N` is the nominal height of messages. Some messages may be forced to use more lines owing to a finite screen width (See option `-width(...)` later in this section).

The default height option is `-hrB^3`.

For `N = 4` the error messages have the general form:

```
File File-name, Line Line-number
Source-line
  I
Error-number: Message
```

where, if the letter '`a`' had been specified, the indicator `I` would have been placed above *Source-line* rather than below.

Example (`-hb^4`):

```
File x.c, Line 4
n = m;
^
Error 40: Undeclared identifier (m)
```

For `N = 3`, the general form is:

```
Source-line
  I
```

*File-name Line-number Error-number: Message*

Message Example (-hb^3):

```
n = m;
^
x.c 4 Error 40: Undeclared identifier (m)
```

Message Example (-hB^3):

```
x.c 4 Error 40: Undeclared identifier (m)
n = m;
^
```

Message Example (-ha\_3):

```
-
n = m;
x.c 4 Error 40: Undeclared identifier (m)
```

For  $N = 2$ , the general form is:

*Source-line*  
*File-name Line-number Error-number: Message*

Example (-h\$2):

```
n = $m;
x.c 4 Error 40: Undeclared identifier (m)
```

For  $N = 1$ , the general form is the same as for  $N = 2$  except the *Source-line* is omitted.

Example (-h1):

```
x.c 4 Error 40: Undeclared identifier (m)
```

**+html**(*sub-option*, ...) emit HTML output

The option **+html** is used when the output is to be read by an HTML browser. An example of the use of this option is shown in the file **env-html.lnt**. That file will enable you to portray the output of PC-lint Plus in your favorite browser.

With this option, lines that echo user source code (as well as lines that contain the horizontal position indicator) are output in a monospace font. New lines are preceded by the HTML escape "<br>". This affects messages and verbosity that are written to standard out. It does not affect verbosity that is also directed to standard error. That is, some verbosity messages are directed to both standard out and to standard error through use of the **+v...** form of the verbosity option. Only the data directed to standard out is affected.

As a reminder, standard out is the normal **stdout** of PC-lint Plus or, if the **-os(filename)** option is given, the destination designated by that option. Standard error is the normal **stderr** or, if the **-oe(filename)** option is given, the destination designated by that option.

The sub-options are:

- **version(html-version)** can be used to designate the version of HTML. Its use is optional. The version identification will be placed within angle brackets and output before the **<html>** at the start of the output file.
- **head(file)** is another optional argument and can be used to supply header information for the HTML output. The *file* is searched for (in the usual places as if it had been specified on a **#include** line) and copied into standard output just after the line that contains "**<html>**" that normally begins an HTML file.

`-limit(n)` set the message limit to *n*

This option imposes an upper limit on the number of messages that will be produced. By default there is no limit.

`++limit(n)` locks in the message limit at *n*

This is a variation of `-limit(n)`. It locks in the limit making it impossible to reverse by a subsequent limit option.

`-message(text)` emits a custom message via info 865

Allows the user to issue a special message with message number 865 and the contents of '*text*' at the time this option is encountered. Environment variables are replaced if surrounded by % characters. For example, you might put this in your `std.lnt` file:

```
-message(INCLUDE is set to: %INCLUDE%)
```

Assuming that INCLUDE is set to `C:\compiler\include`, this would yield an 865 informational message whose text is:

```
INCLUDE is set to: C:\compiler\include
```

While macros are not expanded by `-message` itself, macros may contain embedded lint comments, which may in turn contain a `-message` option that can be used to inspect the value of a provided macro. For example:

```
#define MSG(macro) /*lint -message(The value of #macro is macro) */

#define MAC 100
MSG(MAC)
```

will result in message 865 being emitted with the text:

```
The value of "MAC" is 100
```

`+message([#,] text)` emits a custom message with the specified message #

This option is similar to the `-message` option except that it can be called with 2 arguments, in which case the first argument is a custom message number in the range of 8000-8999. A message with the specified message number and text is emitted.

`+paraminfo(# [,#...])` include parameter information for specified messages as verbosity

`-paraminfo(# [,#...])` exclude parameter information for specified messages as verbosity

For each message number equal to or matching #, this option will cause PC-lint Plus to print verbosity information about each parameter cited in the specified message and the AST node type that the message is anchored to, if any. Example:

```
//lint -w1 +e9272
struct A {
    virtual int foo(int feet);
};

struct B : public A {
    int foo(int meters);
};
```

will elicit the message:

```
note 9272: parameter 1 of function 'B::foo(int)' has different name
than overridden function 'A::foo(int)' ('meters' vs 'feet')
int foo(int meters);
~
supplemental 891: previous declaration is here
virtual int foo(int feet);
~
```

Message 9272 contains several parameters of different types. Using `+paraminfo(9272)` provides the following parameterization information immediately before the message is emitted:

```
Parameter info for next message (9272)
AST Anchor Node Type: none
String parameter 1: '1'
Symbol parameter 1: 'B::foo' (CXXMethodDecl) of type 'int (int)'
Symbol parameter 2: 'A::foo' (CXXMethodDecl) of type 'int (int)'
Symbol parameter 3: 'meters' (ParmVarDecl) of type 'int'
Symbol parameter 4: 'feet' (ParmVarDecl) of type 'int'
```

This can be useful to better understand how messages are parameterized and how suppressions can be applied to specific instances of a message.

The three types of message parameters reported are *String*, *Symbol*, and *Type*. For *Symbol* parameters with a type, this type is also reported. The parameters reported by `+paraminfo` are suitable for use in `-estring`, `-esym`, and `-etype` options. For example, any of the options below will work to suppress this instance of message 9272:

```
-estring(9272, 1)           // the referenced parameter number
-esym(9272, B::foo)        // the first symbol referenced
-esym(9272, A::foo)        // the second symbol referenced
-esym(9272, meters)        // the third symbol referenced
-esym(9272, feet)          // the fourth symbol referenced
-etype(9272, int (int))     // the type of the first two symbols
-etype(9272, int)          // the type of the last two symbols
```

The AST node type of *Symbol* parameters are also included in parentheses following the symbol name. This is useful to determine the node types accessible with the `msgSymbolParam` builtin Query function within `-equery` and `+equery` options.

Some messages are issued with locations anchored to a particular AST node. If the "AST Anchor Node Type" is emitted with a value other than `none`, the corresponding AST node may be accessed via the `currentnode` operator in a Query supplied to a `-equery` or `+equery` option.

`-summary` | `-summary=filename` outputs a message summary at the end of processing, optionally to a file

This option causes a summary of all issued messages to be presented after Global Wrap-up processing. If a filename is specified, the output is sent to the named file. If not, output is sent to the same output stream used for normal Lint messages (that is, the one specified by `-os` or, if no such option was issued, `stdout`).

For each message issued, the summary information consists of the message number (e.g. 1736 for "redundant access specifier"), the number of times that the message was issued, the message type (e.g., "Error", "Warning", etc.) and the message text. This forms a list of all Lint messages that were issued. The list is preceded by a row of column labels to aid readability.

See `-format_summary`.

`-t#` sets tab width

Sets the tab width used for indentation checking. The default tab width is 8, but it can be changed to 4 using `-t4`.

`+typename(# [,#...])` includes the types of symbols when emitting specified messages

`-typename(# [,#...])` excludes the types of symbols when emitting specified messages

For each message number equal to or matching `#`, this option will cause PC-lint Plus to add type information for any and all symbol parameters cited in the specified message. Example:

```
class A{};
void g(A a) {}
// Lint reports "Info 1746: parameter 'a' in function 'g(A)'
// could be made const reference"
//lint +typename(174?)
void f(A a) {}
// Lint reports "Info 1746: parameter 'a' of type 'A' in function 'f(A)'
// of type 'void (A)' could be made const reference"
```

One of the purposes of this option is to show the user an exact type name to use as an argument to `-etype()`. See also `+paraminfo`.

`-width(# [,Indent])` sets the maximum output width and indentation level for continuations

An example of the width option is:

```
-width(99,4)
```

The first parameter specifies the width of messages. Lines greater than this width are broken at blanks. A width of 0 implies no breaking. The second number specifies the indentation to be used on continued lines. `-width(79,4)` is the default setting.

`+xml([tag])` activates XML escape sequences

By adroit use of the `-format` option you may format output messages in xml. See the file `env-xml.lnt` for an example. This option has two purposes. Special xml characters ("`<`", "`>`" "`&`" " " ", and at this writing) will be escaped (to "`&lt;`";, "`&gt;`"; and "`&amp;`";, "`&quot;`";, and "`&apos;`"; respectively) when they appear in the variable portion of the format. Secondly, if `tag` is not null, the entire output will be bracketed with `<tag> ... </tag>`. If `tag` is null this bracketing will not appear.

It is also possible to add a tag in angle brackets that could be used to define, for example, the version of xml. Thus:

```
+xml(?xml version="1.0" ?)
+xml(doc)
```

will produce as a prefix the following two lines.

```
<?xml version="1.0" ?>
<doc>
```

Then, at the end of all the message output the following one line will appear.

```
</doc>
```



## 4.4 Processing Options

### 4.4.1 Compiler Adaptation

`-lang_limit(C|C++ , limit-name, limit-value)` specify minimum language translation limits

The `-lang_limit` option allows customization of the minimum translation limits reported by message [793](#). This option can be used to increase, decrease, or restore the default limits for individual categories when processing C and C++ code.

The first argument is C or C++ indicating which language the change should affect. The second argument is the name of the limit to modify. The third argument is either a non-negative integral value or the special value `default` indicating that any customized value should be removed, restoring the default for the language. For example:

```
-lang_limit(C++, function_arguments, 10)
```

Will cause message [793](#) to be emitted whenever a function call with more than 10 arguments is encountered in a C++ file. A value of 0 for *limit-value* can be used to indicate the lack of a limit. For a list of the supported limits, default values, and limit names for use with this option, see the [17.10 Language Limits](#) section.

`-build_module_interface_unit(filename[, module_decl_name])` build a C++ module interface unit

Build a C++20 module interface unit. The optional second argument must be used to provide the name of the module declaration if the module declaration name differs from the file's base name. A module interface unit must be registered using this option prior to any translation units that import that module interface unit. Building a module interface unit does not automatically subject it to analysis, and non-error messages will not be emitted when building a module interface unit. If a module interface unit is part of your project and its content is within your desired analysis scope, then it should also be listed as a regular source file for analysis in addition to being built using this option for imports.

The addition of C++20 Modules introduces an overlap in terminology between the existing widespread use of the term “module” to refer to C or C++ source files (including frequent use within this documentation) and a reference to C++20 Modules. For clarity, mentions of the bare term “module” throughout the documentation consistently refer to the former sense. References to C++20 Modules functionality are indicated by explicit use of specific terms such as “module interface unit”, “module implementation unit”, “module partition”, “module declaration”, and “C++20 Modules”.

Note that module partitions are not currently supported.

`-std={c89|c90|c99|c11|c17|c23|c++98|c++03|c++11|c++14|c++17|c++20|c++23}` specifies the C or C++ language version

The supported values for this option are:

- C: c89, c90, c99, c11, c17, c23
- C++: c++98, c++03, c++11, c++14, c++17, c++20, c++23

For example, `-std=c99` sets the C language version to C99, and `-std=c++14` sets the C++ language version to C++14.

Note that both the current C language version and the current C++ language version are always stored independently of each other; this option controls which C language version is used when processing a C file and which C++ language version is used when processing a C++ file. The determination of

whether an individual source file is processed as C or C++ is made based on the file extension (see `+cpp`) and the value of the `fcp` flag.

#### 4.4.2 Preprocessor

`-d{Name} [= {Value}]` define the preprocessor symbol *Name* with value *Value*

This option allows the user to define preprocessor variables (and even function-like macros) from the command line. The simplified format of this option is:

```
-dName [=Value]
```

where the square brackets imply that the value portion is optional. If `=Value` is omitted, 1 is assumed. If only *Value* is omitted as in `-dX=` then the value assigned is null. For alternative syntax allowing easier use of string literals in the replacement, use `-dName{Replacement}`. Examples:

```
-dDOS
-dalpha=0
-dX=
```

These three options are equivalent to the statements

```
#define DOS 1
#define alpha 0
#define X
```

appearing at the beginning of each subsequent module.

Note that case is preserved. There is no intrinsic limit to the number of `-d` options. See also the `-u...` option.

This option does not provide any functionality over what can be provided through the use of `#define` within the code. It does allow lint to be customized for particular compilers without modifying source. It also applies globally across all modules, whereas `#define` is local to a specific module.

`+dName [=Value]` define the preprocessor symbol *Name* resistant to change via `#define`

`++dName [=Value]` define the preprocessor symbol *Name* that cannot be `#undef`'d

For added security `++dName=Value` will behave in a similar fashion and, moreover, name cannot be `undef`'ed.

Using this option you can lock in the definition of function-like macros as well as object macros.

For example, suppose the PC-lint Plus is stumbling badly over the macro

```
offsetof(s,m)
```

First place your definition within a header file under a slightly different name:

```
#define my_offsetof(s,m) some_definition...
```

Then use the options:

```
+doffsetof=my_offsetof
-header( my_offsetof.h )
```

where `my_offsetof.h` contains the definition of the `my_offsetof` macro.

You may also explicitly set function-like macros. See `-dName`.

`-header(Filename)` auto-include *Filename* at the beginning of each module

`--header | --header(Filename)` clears previous auto-includes and optionally adds a new one

This is useful for defining macros, **typedefs**, etc. of a global nature used by all modules processed by PC-lint Plus without disturbing any source code. For example,

```
-header( lintdcls.h )
```

will cause the file `lintdcls.h` to be processed before each module.

The header is not reported as being unused in any given module (even though it may be). It is not considered a library header. An extra option may be needed to make this assertion as follows:  
`+libh(lintdcls.h)`

Multiple `-header` options may be used, and this effect is additive. Files are included in the order in which they are given. However, an option of the form:

```
--header( Filename )
```

will remove **all** prior headers specified by `-header` options before adding *Filename*.

If *Filename* is absent as in `--header` then the effect is to erase **all** prior `-header` requests.

`-idirectory` add search *directory* for `#include` directives

`-Idirectory` add search *directory* for `#include` directives

Files not found in the current directory are searched for in the directory specified. There is no intrinsic limit to the number of such directories. The search order is given by the order of appearance of the `-idirectory` options. For example:

```
-i/lib/
```

can be used to make sure that all files not found in the current directory are looked up in some library directory named `lib`. A directory separation character will be appended onto the end of the `-i` option if not already present. Thus

```
-i/lib
```

is equivalent to the above.

To include blanks within the directory name employ the quote convention (See Section 4.1 Rules for Specifying Options) as in the following:

```
-i"program files\compiler"
```

Multiple directories may be specified either with multiple `-i` options or by specifying a semi-colon separated list with a single `-i` option. (See also `+fim`)

PC-lint Plus also supports the INCLUDE environment variable, See Section 18.2.1 INCLUDE Environment Variable. Note: Any directory specified by a `-i` directive takes precedence over the directories specified via the INCLUDE environment variable.

`-Idirectory` is identical to `-idirectory`.

As a special case the option `-i-` is taken as a directive to remove all of the directories established with previous `-i` options (it has no effect on those directories specified with INCLUDE).

`--idirectory` add lower-priority search *directory* for `#include` directives

All directories specified by `-i` are searched before directories named by `--i`. This is to support compilers that always search through compiler-provided library header directories after searching user-provided directories.

Example: suppose there is a header file named `'bar.h'` in both directory `'/foo'` and directory `'local'`. Then:

```
// in std.lnt:
--i/foo      // search foo with low priority
-ilocal     // search local with high priority
// in t.cpp:
#include <bar.h> // finds the version in 'local'
```

Thus the priority of the `--i` option is always lower than the `-i` option. How does it compare with the `INCLUDE` environment variable, which is also lower than `-i`? If the `--i` option is in an indirect file (`.lnt` file) it will act as though it had a lower priority than the `INCLUDE` environment variable. This is because the `INCLUDE` variable is triggered upon reading the first file. If the `--i` option is on the command line or in the `LINT` environment variable it will take priority over the `INCLUDE` variable.

`-incvar(Name)` change the name of the `INCLUDE` environment variable to *Name*

The environment variable `INCLUDE` is normally checked for a list of directories to be searched for header files (See Section [18.2 include Processing](#)). You may use the `-incvar(Name)` option to specify a variable to be used instead of `INCLUDE`. For example

```
-incvar(MYINCLUDE)
```

requests checking the environment variable `MYINCLUDE` rather than checking `INCLUDE`.

Limitation: This option may not be placed in an indirect `.lnt` file or source file. It may be placed on the command line or within the `LINT` environment variable. The `INCLUDE` environment variable is processed just before opening the first file.

`-pch(Header)` designates a given header as the pre-compiled header, creating precompiled form if needed

`+pch(Header)` designates a given header as the pre-compiled header, forcing recreation

The *header name* should be that name used between angle brackets or between quotes on the `#include` line. In particular, if the name on the `#include` line is not a full path name do not use a full path name in the option. See Section [6.2 Designating the precompiler header](#).

`-pp_sizeof(Text, Value)` set the value that `sizeof(Text)` evaluates to in a preprocessor directive

This option is provided for legacy code and will direct PC-lint Plus on how to evaluate a particular `sizeof` expression appearing in a preprocessor conditional. See [18.6 Preprocessor sizeof](#).

`-uName` undefine the preprocessor symbol *Name*

For example:

```
-u_lint
```

will undefine the identifier `_lint`, which is normally pre-defined before each module. The undefine will take place for all subsequent modules after the default pre-definitions are established. The observant reader will notice that you may not undefine the name `nreachable`.

`--uName` ignore past and future `#defines` of the preprocessor symbol *Name*

```
//lint --uX
#define X 1
int y = X;
```

will be equivalent to:

```
int y = X;
```

Please note the difference between this option and the `-uName` option, which undefines any built-in definition for *Name* but does not affect definitions that *Name* may acquire in the future.

```
+ppw(word [,word...]) enable preprocessor keyword(s)
-ppw(word [,word...]) disable preprocessor keyword(s)
```

This option removes any predefined meaning we may associate with the preprocessor word(s) (*Word1*, *Word2* etc.). If this is followed by a `+ppw(word)` the word is entered as a no-op rather than one that has a predefined meaning. For example, if your code contains the non-standard preprocessor directive `#include_next` and if its meaning coincides with that of the GNU compiler, then just issue the option `+ppw(include_next)`. However, if you would rather have it ignore such a preprocessor word, issue the commands:

```
--ppw(include_next)
+ppw(include_next)
```

```
--ppw(word [,word...]) remove built in meaning of preprocessor keyword(s)
```

If this is followed by a `+ppw(word)` the word is entered as a no-op rather than one that has a predefined meaning. For example, if your code contains the non-standard preprocessor directive `#dictionary` and if its meaning coincides with that of the DEC VMS compiler, then just issue the option `+ppw(dictionary)`. However, if you would rather have it ignore such a preprocessor word, issue the commands:

```
--ppw(dictionary)
+ppw(dictionary)
```

```
-ppw_asgn(Word1, Word2) assign preprocessor word meaning of Word2 to Word1
```

This option assigns the preprocessor semantics associated with *Word2* to *Word1* and activates *Word1*. E.g.

```
-ppw_asgn( header, include )
```

will then make

```
#header <stdio.h>
```

behave exactly like:

```
#include <stdio.h>
```

The purpose of this option is to support special non-standard preprocessor conventions provided by some given compiler.

Even though *Word2* may not be activated it may still have semantics. Thus

```
-ppw_asgn( INC_NEXT, include_next )
```

will assign the semantics associated with the `include_next` preprocessor directive to `INC_NEXT` and activate `INC_NEXT`; all this in spite of the fact that `include_next` has not been activated (with the `+ppw` option). See Section [18.4 Non-Standard Preprocessing](#) for descriptions of non-standard

preprocessing directives.

The `#macro` pre-processing directive is not a directive implemented by any compiler to our knowledge. So why, you ask, are we providing this directive? We are providing `#macro` as a way for programmers to implement arbitrary preprocessor directives by converting directives into macros.

For example, one compiler accepts the following preprocessor directive

```
#BYTE n = 'a'
```

as a declaration of the variable `n` having a type of `BYTE` and an initial value of `'a'`. The `#macro` directive will allow us to express `#BYTE` as a macro and so render the directive as C/C++ code.

The preprocessing directive:

```
#macro a b c
```

will result in the macro invocation

```
sharp_macro( a b c )
```

The word `sharp` is used as a prefix because the word `'sharp'` is often used to denote verbally the `'#'` character.

We can transfer the properties of `#macro` to some other actual or potential preprocessing directive using the option `ppw_asgn`. For example:

```
-ppw_asgn( BYTE, macro )
```

will assign the `#macro` properties to `BYTE` (and also enable `BYTE` as a preprocessing directive). Then the directive

```
#BYTE n = 'a'
```

will result in the macro call:

```
sharp_BYTE( n = 'a' )
```

Presumably there is a macro definition that resembles:

```
#define sharp_BYTE(s) unsigned char s;
```

Such a definition can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

`+pragma(Identifier, Action)` associates *Action* with *Identifier* for `#pragma`  
`-pragma(Identifier)` disables pragma *Identifier*

The `+pragma(identifier, Action)` option can be used to specify an *identifier* that will be used to trigger an *action* when the *identifier* appears as the first identifier of a `#pragma` statement. See Section [4.12.7 User pragmas](#).

### 4.4.3 Tokenizing

`-$ permit $` in identifiers

`-ident(chars)` add identifier characters

This option allows the user to specify alternate identifier characters. Each character in *chars* is taken to be an identifier character. For example if your compiler allows `@` as an identifier character then you may want to use the option:

`-ident(@)`

Option `-$` is identical in effect to `-ident($)` and is retained for historical reasons.

`+rw(word [,word...])` enable reserved word(s)

`-rw(word [,word...])` disable reserved word(s)

If the meaning of a reserved word being added is already known, that meaning is assumed. For example, `+rw(typeof)` will enable the reserved word `typeof`. If the reserved word has no prior known semantics, then it will be passed over when encountered in the source text. As another example:

`+rw( __inline, entry )`

adds the two reserved words shown. `__inline` is assigned a meaning consistent with that of the Microsoft C/C++ compiler. `entry` is assigned no meaning; it is simply skipped over when encountered in a source statement. Since no meaning is to be ascribed to `entry`, it could just as well have been assigned a null value as in

`-dentry=`

`--rw(word [,word...])` remove built in meaning of reserved word(s)

If *word* has known semantics, remove those semantics. For example, `+rw(global)` installs the reserved word `global` with the meaning it has in OpenCL. If you don't want that meaning but would rather have `global` ignored, then use the option sequence:

`--rw(global)`

`+rw(global)`

`-rw_asgn(Word1, Word2)` assigns reserved word meaning of *Word2* to *Word1*

assigns the keyword semantics associated with *word2* to *word1* and activates *word1*. E.g.

`-rw_asgn( interrupt, _to_brackets )`

will assign the semantics of `_to_brackets` to `interrupt`. This will have the effect of ignoring `interrupt(21)` in the following:

`void f( int n ) interrupt(21) { }`

The purpose of this option is to support special non-standard keyword conventions provided by some given compiler. But do not overlook the use of the `-d` option in this connection. `-d` (or the equivalent `#define`) can be more flexible since a number of tokens may be associated with a given identifier.

#### 4.4.4 Parsing

`-ar_limit=n` set the operator arrow depth limit to *n*

This option specifies the *arrow limit* which represents the maximum depth of which calls to `operator->` functions are processed. The default value for this option is 256.

`-br_limit=n` set the bracket depth limit to *n*

This option specifies the *bracket limit*. When a nesting of parentheses, square brackets, and curly braces exceeds this limit an error will be emitted. The default value for this option is 256.

**-cc\_limit=*n*** set the `constexpr` call depth limit to *n*

This option specifies the *constexpr call limit* which represents the maximum call depth of compile-time-evaluated `constexpr` functions. The default value for this option is 512.

**-cs\_limit=*n*** set the `constexpr` step limit to *n*

This option specifies the *constexpr step limit* which represents the maximum number of evaluation steps performed during compile-time-evaluated `constexpr` functions. The default value for this option is 1048576. Exceeding this limit may be the result of an infinite loop in a `constexpr` function.

**-fallthrough** ignores switch case fallthrough when used in a lint comment

indicates that the programmer is aware of the fact that flow of control is falling through from one case (or default) of a switch to another. Without such an option Message 825 will be issued. For example:

```
case 1:
case 2: n=0;      //setting n to 0
case 3: n++;
```

will result in Info 825 on case 3 because control is falling through from the statement above, which is neither a case nor a default. The cure is to use the **-fallthrough** option:

```
case 1:
case 2: n = 0;    //setting n to 0
//lint -fallthrough
case 3: n++;
```

Warning 616 will be issued if no comment at all appears between the two cases. If this is adequate protection, then just inhibit message 825.

**-tr\_limit=*n*** set the template recursion limit to *n*

This option specifies the *template recursion limit*. When the limit is reached, message 1777 is issued, which reminds you that you may use this option to deepen the level of recursion. See message 1777 for further details. The default value for this option is 1024.

**-unreachable** ignores unreachable code when used in a lint comment

This is useful to inhibit some error messages. For example, suppose `my_exit()` does not return. Then:

```
int f(n)
{
    if(n) return n;
    my_exit(1);
    //lint -unreachable
}
```

contains an unreachable indicator to prevent PC-lint Plus from thinking that an implied return exists at the end of the function. An implied return would not return a value but `f()` is declared as returning `int`. Note, however, that it would have been better practice to copy the exit semantics to `my_exit`. Eg.:

```
-function(exit,my_exit)
```

In this case the **-unreachable** option would not have been necessary.



## 4.5 Data Options

### 4.5.1 Scalar Data Size

**-s** set the size of various types

This option allows setting the size of various scalars (**short**, **float**, pointers, etc.) for the target machine. The default sizes are for a 32-bit target machine following the ILP32 model. If you are targeting a 32-bit system that uses a different data model, targeting a 64-bit architecture, or targeting an embedded environment, then you will need to adjust type sizes to match.

For example, an LP64 target would use the size options:

```
-ss2 -si4 -sl8 -sll8 -sp8
```

and an embedded system with 16-bit characters might use:

```
-sb16 -ss1 -si1 -sl1 -sll2 -sp2
```

Note that the maximum size that can be specified for standard types is 64 bits. Because the size of **long long** defaults to 8 bytes, any increase in the size of a byte will require a decrease in the size of **long long** or a size option misconfiguration error will be issued for **long long**.

In the list below **#** denotes a small positive integer that represents the size in characters for the corresponding type (except for **-sb#** where **#** is in units of bits).

- sb#**    The number of bits in a **char** is **#**. The default is 8. Note that by the definition of the **sizeof** operator, **sizeof(char) == 1** regardless of **#**. This value is conceptually similar to the standard macro **CHAR\_BIT**, although your standard library must still define this correctly in a way that matches the value set using this option.
- sbo#**    **sizeof(bool)** becomes **#**. The default is 1.
- ss#**    **sizeof(short)** becomes **#**. The default is 2.
- si#**    **sizeof(int)** becomes **#**. The default is 4.
- sl#**    **sizeof(long)** becomes **#**. The default is 4.
- sll#**    specifies the size of **long long**. The option **-sll#** can be used to specify the size of a **long long int**. The default is 8. Specifying this size also enables the flag **+fll**.
- sf#**    **sizeof(float)** becomes **#**. The default is 4.
- sd#**    **sizeof(double)** becomes **#**. The default is 8.
- sld#**    **sizeof(long double)** becomes **#**. The default is 8.
- sp#**    size of pointers become **#**. The default is 4. There must be at least one integer type size that matches the pointer size.
- sw#**    **sizeof(wchar\_t)** becomes **#**. The default is 2. Note this only affects a built-in **wchar\_t**.

See **-a** to specify alignment of types and the relationship between size and alignment.

**-size(flags, amount)** set static or auto size thresholds

This option causes an Informational message (812, 813, 2712 or 2713) to be issued whenever a data variable's size in bytes equals or exceeds a given *amount*. *Flags* can be used to indicate the kind of data as follows:

- s** static data (Info 812). Data can be file scope (**extern** or **static**) or declared **static** and local to a function.
- a** auto data (i.e., stack data) (Info 813). See also **-stack**, which can do a comprehensive stack analysis.
- p** parameter variable (Info 2712). Checks function parameters for types of a size that equal or exceed the given amount.
- r** return value (Info 2713). Checks for functions that return a type with a size that equal or exceed the given amount.

The purpose of the **-size** option is to detect potential causes of stack overflow (using the '**a**' flag); to flag large contributors to excessively large static data areas (using the '**s**' option); to identify functions that are receiving (using the '**p**' option) or returning (using the '**r**' option) large data types that may be more appropriate to pass by pointer or reference.

E.g. **-size(a,100)** detects auto variables that equal or exceed 100 bytes. If you have a stack overflow problem, such a test will let you focus on a handful of functions that may be causing the overflow. It does not, however, look at call chains and does not compute an overall stack requirement either of a single function or of a sequence of calls.

If *amount* is 0 (it is by default) no message is given.

#### 4.5.2 Scalar Data Alignment

**-a** set the alignment of various types

The address at which an object may be allocated must be evenly divisible by its type's alignment. For example, if the type **int** has an alignment of 2 then each **int** must be allocated on an even byte address.

Alignment has only minimal impact on the behavior of PC-lint Plus. At this writing there are only three messages (958, 959 and 2445) that detect alignment irregularities. But in addition to these it is sometimes essential to get alignment correct in order to know the sizes of compound data structures.

For every size option (see **-s**) of the form:

**-sType#**

(except for **-sb#**, the byte size) there is an equivalent alignment option having the form:

**-aType#**

For example, the option

**-ai1**

indicates that the alignment of **int** is 1 byte. An alignment of 1 means that no restriction is placed on the alignment of types. (An alignment of 0 is undefined).

If an alignment for a type is not explicitly given by a **-a** option, an alignment is deduced from the size of the type. The deduced alignment is the largest power of 2 that evenly divides the type. Thus if the size is 4 the deduced alignment is 4 but if the size is 6 the deduced alignment is 2. This deduction is made just before the first time that alignment (and size) may be needed. An attempt to use the **-aType** option (or the **-sType** option) after this point is greeted with Error 686. For example:

```
-si8      // sizeof(int) becomes 8
          // alignment of int also becomes 8
```

```

-ai1      // alignment of int is 8
-si16     // sizeof(int) becomes 16
          // alignment of int stays at 1
-sl24     // sizeof(long) is 24
          // alignment of long is 8
-al2      // alignment of long becomes 2

```

## 4.6 Miscellaneous Options

### 4.6.1 File

`+cpp(Extension [,Extension...])` add C++ extension(s)  
`-cpp(Extension [,Extension...])` remove C++ extension(s)

This option allows the user to add and/or remove extensions from the list that identifies C++ modules. By default only `.cpp` and `.cxx` are recognized as C++ extensions. (It is as if `+cpp(cpp,cxx)` had been issued at the start of processing.) For example:

```
lint a.cpp +cpp(cc) b.cc c.c
```

treats `a.cpp` and `b.cc` as C++ modules and `c.c` as a C module. There is no intrinsic limit to the number of different extensions that can be used to designate C++ modules. See also flag `+fcp`.

Note: If you are using `+cpp(.C)`, i.e. you want to use case to distinguish C++ vs. C on Windows, you need to also turn off the fold file name flag (`-fff`).

`+ext(Extension [,Extension...])` set the extensions to try for extensionless files

For example,

```
lint alpha
```

will, by default, cause first an attempt to open `alpha.lnt`. If this fails there will be an attempt to open `alpha.cpp`. If this fails there will be an attempt to open `alpha.cxx`. Finally, an attempt to open `alpha.c` will be made. It is as if the option:

```
+ext( lnt, cpp, cxx, c )
```

had been given on startup.

Minor notes: This has no effect on which extensions indicate that a module is to be regarded as a C++ module. This is done by the options `-/+cpp` and `-/+fcp`. Prefixing an extension with a period has no effect. Thus, `+ext(lnt,c)` means the same as `+ext(.lnt,.c)`. For Windows, upper-casing the extension also has no effect. Thus, `+ext(lnt)` has the same effect as `+ext(LNT)`. On Unix, however, case differences do matter. For example, if the Unix programmer wanted both `.c` and `.C` extensions to be taken by default he might want to use the option: `+ext(lnt,c,C)`.

`+headerwarn(Filename)` causes message #829 to be issued when *Filename* is `#included`

For example `+headerwarn(stdio.h)` will alert the programmer to the use of `stdio.h`. If the option `-wlib(1)` is in place, as it usually is to stem the flood of Warnings and Informationals emanating from library headers, no message 829 will be issued from within a library header unless you also issue a `+elib(829)` sometime after the `-wlib(1)`.

`-indirect(File [,...])` process *File* as an options file

Allows you to specify Lint option files to be processed when this option is encountered. This is useful if you want to use an options file within a Lint comment. For example, Lint option files that are appropriate only for a particular configuration may be conditionally included. The code below may appear in some header that is included either in every module or at least the first module. Thus, the header in question can be injected with the `-header` option.

```
#ifndef BEEN_HERE
#define BEEN_HERE
    #if defined(HAS_LIBRARY_A)
        //lint -indirect(lib-a.lnt)
    #elif defined(HAS_LIBRARY_B)
        //lint -indirect(lib-b.lnt)
    #else
        //lint -indirect(lib-default.lnt)
    #endif
#endif
```

`+libclass(Identifier [...])` add class of headers treated as libraries

This option specifies the class of header files that are by default treated as library headers. Arguments can be one of:

- `angle` (specified with angle brackets),
- `foreign` (comes from a foreign directory using `-i` or the `INCLUDE` environment variable),
- `ansi` (one of those specified by ANSI/ISO C), or
- `all` (meaning all header files).

For more information, see Section [5.1 Library Header Files](#).

`+libdir(Directory [...])` specify a *Directory* of headers to treat as libraries

`-libdir(Directory [...])` specify a *Directory* of headers to not treat as libraries

This option allows you to override `+libclass` for specified directories. *Directory* may contain wild cards ('\*' and '?'). For more information, see Section [5.1 Library Header Files](#).

`+libh(Header [...])` specify *Headers* to treat as libraries

`-libh(Header [...])` specify *Headers* to not treat as libraries

This option allows you to override `+libclass` and `+/-libdir` for specified headers. *Header* may contain wild card characters. For more information, see Section [5.1 Library Header Files](#).

`+libm(Module [...])` specify *Modules* to treat as libraries

`-libm(Module [...])` specify *Modules* to not treat as libraries

The *Module* may contain wild card characters. For more information, see [5.2 Library Modules](#).

`-library` indicates the next source module is to be treated as library code

This option turns ON the library flag for the next module, if given on the command line, or for the rest of the module if placed within a lint comment. For an example, see Section [5.2 Library Modules](#). At one time this option was equivalent to the `+flb`. However, there is now a difference. `-library` designates the file in which it is placed and all files that it may include as having the library property. Thus, if a lint option within a header file contains the `-library` flag then only that header (and the headers it includes) are affected. It does not affect the including file. With `+flb` the flag is left on

until turned off.

```
+lnt(Extension [,Extension...]) add indirect file extension(s)
-lnt(Extension [,Extension...]) remove indirect file extension(s)
```

Modify the list of filename extensions used to indicate indirect files (by default only `lnt` designates an indirect file). For example, if you want files ending in `.lin` to be interpreted as indirect files you use the option:

```
+lnt( lin )
```

After such an option, a filename such as `alpha.lin` will be interpreted as if it had been named `alpha.lnt`. That is, it will be interpreted as an extension of the command line rather than as a C/C++ program.

This will not affect the sequence of default extensions that are tried. Thus, when the name `alpha` is encountered, there will not first be a test to see if `alpha.lin` exists. This is governed by the `+ext` option.

If you want to remove the name `lnt` and replace it by `lin` you need to use the pair of options:

```
-lnt(lnt) +lnt(lin)
```

```
-astquery register a Query to be evaluated during AST traversal
```

Specifies a Query that will be evaluated for visited nodes during traversal of module ASTs. This option may be used to implement custom checks and messages. See Chapter 16 [Queries](#) for more information.

```
-dump_queries disable dumping of parsed Query ASTs
+dump_queries([sub-options]) enable dumping of parsed Query ASTs
```

Enables or disables AST dumping for subsequently parsed Queries, see Section 16.7.2 [Dumping the Query Tree](#) for more information.

#### 4.6.2 Global

```
? displays help
```

```
-b suppress banner output
+b redirect banner output to stdout
++b produce banner line
```

(Unlike most other options, this option must be placed on the command line and not in an indirect file.) When PC-lint Plus is run from some environments, the banner line (identifying the version of PC-lint Plus and bearing a Copyright Notice) may overwrite a portion of an editing screen. This is because the banner line is, by default, written to standard error whereas the messages are written to standard out and can be redirected. The option `+b` will cause the banner line to be written to standard out (and hence will become part of the redirected output). The option `-b` will suppress the banner line completely.

The option `+b` works well for:

```
lint +b ... >outfile
```

Unfortunately this will not have the intended effect with:

```
lint +b -os(outfile) ...
```

as the banner line is written before the `-os` option has had a chance to take effect.

`++b` will deposit the banner line into standard out anywhere it is encountered. Thus:

```
lint -os(outfile) ++b ...
```

will cause the banner line to be placed into `outfile`. You will also get a banner line in standard error but this can be separately suppressed as in:

```
lint -b -os(outfile) ++b ...
```

`-cond(conditional-expr, true-options [, false-options])` conditionally execute options

The `-cond` option accepts two or three arguments, the first of which is a conditional expression to evaluate when the option is processed, the second is the set of options to execute if the conditional expression evaluates to true, and the third (optional) argument specifies the options to execute if the conditional expression is false.

The conditional expression may contain string and numeric comparisons and string pattern matching using regular expressions. Each operand in an expression is either numeric or a string. A string is any text that is surrounded by single quotes, everything else is numeric. Valid numeric values include anything that the standard C function `strtod` can parse as well as the literal values `true` and `false`, which represent the values 1 and 0 respectively.

The arithmetic operators `+`, `-`, `*`, and `/` may be used on numeric values within the expression and possess the same meaning and precedence as their corresponding C operators. The `/` operator performs floating point division, e.g. `1 / 2` will evaluate to 0.5, not 0. The `//` operator performs integer division and has the same precedence as `/`. The `mod` operator performs integer modulus arithmetic and is equivalent to the `%` operator in C. The `fmod` operator yields the floating point modulus value of its operands. The unary operators `+`, `-`, and `!` have the traditional meaning when applied to numeric operands.

The comparison operators `<`, `<=`, `>`, `>=` and the equality operators `==` and `!=` can be used on either numeric or string operands. When used with numeric operands, they behave as the corresponding C operators. When used with string operands, they perform lexicographic comparisons. The operands must be of the same type (numeric or string).

The logical operators `&&` and `||` and the ternary operator `?:` are supported and have the same meaning as the corresponding C operators. Parentheses may be used for grouping subexpressions.

The pattern matching operator `~` takes two operands, a subject string on the left-hand side and a pattern on the right-hand side. Both operands must be strings. The result is true if the subject string matches the regular expression pattern and false otherwise. The PCRE (Perl) regular expression syntax is used for the regular expressions supported by the `-cond` option.

The special identifier `__is_stdout_terminal` evaluates to 1 if standard output does not appear to have been redirected or piped and 0 otherwise.

The `-cond` option is often used to conditionally execute options based on the value of defined environment variables. For example, the option:

```
-cond('%USE_EXTRA%' == '1', extra.lnt)
```

will cause PC-lint Plus to process the file `extra.lnt` if the environment variable `USE_EXTRA` is defined with a value of 1.

In the following example, the `WARNING_LEVEL` environment variable is always expected to be defined with a value between 1 and 4. If the variable is properly defined, it is used to set the default warning level, otherwise a fatal error is emitted to prevent further processing:

```
-cond('%WARNING_LEVEL%' ~ '[1234]$',
      -w%WARNING_LEVEL%,
      +fatal_error("WARNING_LEVEL environment variable not properly defined"))
```

`-dump_messages(file=filename [,format={plain|list|json|yaml|csv|xml}] [,sub-options])`  
 dumps PC-lint Plus messages to the provided file in the specified format

This option writes out the PC-lint Plus message list to the filename specified with the `file=filename` sub-option. The `format` sub-option specifies the format to use when writing messages and may be any of `plain`, `list`, `json`, `yaml`, `csv`, or `xml`. Specifying a format of `json`, `yaml`, `csv`, or `xml` will result in messages being written in the corresponding format (see examples below). A format of `plain` will result in output similar to the `msg.txt` file provided with PC-lint 9. A format of `list` results in a list output containing one message per line with fields separated by tabs. The default format is `plain`.

The `include_commentary` sub-option may be used to indicate whether Reference Manual descriptions of each message should be included in the output. The default is to include commentary; `include_commentary=false` will disable commentary. Note that commentary is not supported for the `list` format and cannot be disabled for the `plain` format.

The `include_clang_errors` sub-option indicates whether messages in the `4xxx`, `5xxx`, and `6xxx` ranges are included in the output. These messages are mapped clang errors that do not contain descriptions and are excluded from the output by default. Use `include_clang_errors` or `include_clang_errors=true` to include these messages.

In all cases, messages are written in ascending order of message number.

### Examples

`-dump_messages(file=msg.txt)` will result in output that looks like:

```
error 1
unclosed comment

      End of file was reached with an open comment still unclosed.

error 2
unclosed quote

      An end of line was reached and a matching quote character (single or
      double) to an earlier quote character on the same line was not found.
```

`-dump_messages(file=msg.txt,format=list)` looks like:

```
1      error  unclosed comment
2      error  unclosed quote
3      error  #elif without a #if
5      error  too many #endif directives
8      error  unclosed #if
9      error  #elif after #else
```

`-dump_messages(file=msg.txt,format=json)` looks like:

```
[
  {
```

```

        "ID" : "1",
        "CATEGORY" : "error",
        "TEXT" : "unclosed comment",
        "COMMENTARY" : "End of file was reached with an open comment still unclosed."
    },
    ...
]

```

`-dump_messages(file=msg.txt,format=yaml)` looks like:

```

-
  id: 1
  category: error
  text: 'unclosed comment'
  commentary: |
    End of file was reached with an open comment still unclosed.
-
  id: 2
  category: error
  text: 'unclosed quote'
  commentary: |
    An end of line was reached and a matching quote character (single or
    double) to an earlier quote character on the same line was not found.

```

`-dump_messages(file=msg.txt,format=csv)` looks like:

```

"1","error","unclosed comment","End of file was reached with an open comment still unclosed."
"2","error","unclosed quote","An end of line was reached and a matching quote character (single or
double) to an earlier quote character on the same line was not found."
"3","error","#elif without a #if","A #else was encountered not in the scope of a #if, #ifdef or #ifndef."

```

Note that newlines may be embedded in the commentary field.

`-dump_messages(file=msg.txt,format=xml)` looks like:

```

<messages>
  <message id="1">
    <category>error</category>
    <text>unclosed comment</text>
    <commentary>End of file was reached with an open comment still unclosed.</commentary>
  </message>
  ...
</messages>

```

`-dump_message_list=filename` dumps PC-lint Plus message list to the provided file

`-dump_message_list` will cause PC-lint Plus to write out its list of messages to the provided file. For example, `-dump_message_list(mlist.txt)` will write the message information for all messages supported by PC-lint Plus to a file named `mlist.txt`. This file contains one line per message with three fields, delimited by tabs as shown below:

```

25  error   character constant too long for its type
29  error   duplicated type-specifier, '__detail__'
31  error   redefinition of symbol __symbol__
32  error   field size (member __symbol__) should not be zero

```

Parameterized messages show up in the same way that they do when using the `-summary` option.

`-exitcode=n` set the exit code to *n*



By default, PC-lint Plus terminates with an exit code of 0 upon successful completion and an exit code of 1 when terminating prematurely, such as from a fatal error. If the **frz** flag is turned OFF, PC-lint Plus will instead exit with the total number of messages emitted. This option can be used to specify the exit code that PC-lint Plus should return upon completion (successful or otherwise). This option has no effect if the **frz** flag is ON. See also **-zero**, and **+zero\_err**.

**+f** turns a flag on  
**-f** turns a flag off  
**++f** increments a flag  
**--f** decrements a flag

See Section [4.11 Flag Options](#).

**-fatal\_error(message)** triggers a suppressible fatal error using message 398

Issues fatal error [398](#) with the provided text. While a fatal error normally causes execution to terminate immediately, this does not occur if the message is suppressed. Message 398 is one of the few fatal errors for which suppression is permitted.

**+fatal\_error(message)** triggers an unsuppressible fatal error using message 399

Issues fatal error [399](#) with the provided text. A fatal error causes execution to terminate immediately. Message 399 cannot be suppressed.

**-help=Option** display detailed help about *Option*

With no arguments, the **-help** option produces the same output as the **?** option. When provided with the name of an option, help information specific to the provided option is emitted. For example, given **-help=+libh**, the following will be emitted:

```
OPTION:    +libh
GROUP:     Miscellaneous
CATEGORY:  File
USAGE:     +libh(Header [...])
```

specify Headers to treat as libraries

**-max\_threads=n** set the maximum number of concurrent threads for parallel analysis

The **-max\_threads** option can be used to specify the number of concurrent threads (the default is 1, which essentially disables multi-threading). One thread will be created for each source module, up to the specified maximum. The **-max\_threads** option must appear before the first module to have any effect. See Section [17.9 Parallel Analysis](#) for more information.

**-p** | **-p(width)** just preprocess

If this flag is set, the entire character of PC-lint Plus is changed from a diagnostic tool to a preprocessor. The output is directed to standard out, which may be redirected. Thus,

```
lint -os( file.p ) -p file.c
```

will produce in **file.p** the text of **file.c** after all **#** directives are carried out and removed. This may be used for debugging to determine exactly what transformations are being applied by PC-lint Plus.

The optional argument (*width*) denotes an upper bound on the width of the output lines. For example:

`-p(100)`

will limit the width of output lines to 100 characters. Splitting is done only at token boundaries. Very large tokens will not be split even if they exceed the nominal line limit. This is so the result can be passed back through lint or some other C/C++ source processor.

In order to track down some complicated cases involving many include headers you may want to use the `-v1` verbosity option in connection with `-p`. Recall (Section 4.3.2 [Verbosity](#)) that `-v1` will produce a line of output for every line processed. When you use both options together, as in, for example:

```
lint -os( file.p ) -v1 -p file.c
```

then the single line will be preceded by the name of the file and the line number both enclosed in a C comment. This will enable you to track through every line of every header processed.

`-setenv(name=value)` set environment variable *name* to *value*

will allow the user to set an environment string. The directive is of the form *name=value*. For example:

```
-setenv(ROOT_DIR=\home\program\dev)
```

will set the environment variable `ROOT_DIR` to the indicated directory. This can be used subsequently in PC-lint Plus options by using the `%var%` syntax. For example:

```
-i%ROOT_DIR%\include
```

establishes a new search directory based on the environment name.

The environment variable setting will last for the duration of the process. See Section 4.1.11 [Expansion of Environment Variables in Options](#) for more information.

`-skip_function(Function [,Function...])` skips the body of a *Function* when parsing

Causes the bodies of the named functions to be skipped during parsing. Functions whose bodies are skipped cannot be semantically analyzed. This option is useful if you are using compiler-specific syntax that cannot be easily accommodated by PC-lint Plus and such usage is isolated to a handful of functions. In general it is better to configure PC-lint Plus to appropriately handle compiler-specific peculiarities but this option can be used as a last resort. See also the `flf` flag to automatically skip bodies of library functions.

`-subfile(File, options|modules)` process just options or just modules from options file *File*

This is an unusual option and is meant for front-ends trying to achieve some special effect. There are two forms of the option; one with the second argument equal to `options` and the other with the second argument equal to `modules`. In general, indirect files (those ending in `.lnt`) will contain both options and modules. Sometimes it is important to extract just the options from such a file. One example is if you are attempting to do a unit-check on one particular module. Say your project file is `project.lnt`. Then you might do project and unit checks using the same indirect file.

```
lint project.lnt // project check
lint -subfile( project.lnt, options ) filename // unit check
```

Note that `project.lnt` may itself have indirect files and that modules and options may be interspersed. The rule is that every indirect file is followed for as long as it takes until the first module is encountered. Every option thereafter is considered not a general option but specific to project check out.

With `modules` as the second argument to `subfile`, the processing picks up at precisely the point that the 'options' subargument left off. Thus if you wanted to place a particular option, say `-e1706`, just before the first module of `project.lnt` you could achieve that effect by placing the following in either an indirect file or on the command line:

```
-subfile( project.lnt, options ) -e1706
-subfile( project.lnt, modules )
```

**-unit\_check** unit checkout

This is one of the more frequently used options. It is used when linting a subset (frequently just one) of the modules comprising a program and suppresses Global Wrapup and the messages that would be issued during global wrapup analysis. For historical reasons, **-u** is an alias for this option.

**--unit\_check** unit checkout and ignore modules in lower .lnt files

This option is like **-unit\_check** except that any module at a lower .lnt level is ignored. Suppose, for example, that **project.lnt** is a project file containing both options and module names. Then the command line:

```
lint --unit_check project.lnt alpha.cpp
```

will do a unit check on module **alpha.cpp**. It will ignore any module names that may be identified within **project.lnt**. **project.lnt** does not have to immediately follow the **--unit\_check** option. Any .lnt file within **project.lnt** will similarly be processed for options but module names will be ignored. See also **-subfile()** which deals with this issue in a more comprehensive manner.

For historical reasons, **--u** is an alias for this option.

**-write\_file(String, Filename [,append=true|false] [,binary=true|false])** write *String* to file *Filename*

The **-write\_file** option is used to write data to a file. The option accepts a string to write and the name of the file to write the contents of the string to. Backslash escapes appearing in the string are converted as expected. By default, the output file is overwritten, to append data to the end of the file, use the sub-option **append=true**. The sub-option **binary** can be used to write the data in binary mode. For example, to append a line containing the text "Starting Lint version X" where X is the major version number to a file named **/home/pclint/lint.log**, the below is used:

```
-write_file("Starting Lint version %LINT_VERSION%\n",
            "/home/pclint/lint.log", append=true)
```

**-zero** | **-zero(#)** sets exit code to 0

This is useful to prohibit the premature termination of **make** files.

**-zero(#)** will set the exit code to zero if all reported errors are numbered **#** or higher after subtracting off 1000 if necessary. More precisely, messages that have a message number whose modulus 1000 is equal to or greater than **#**, do not increment the error count reported by the exit code. Note that suppressed errors also have no effect on the exit code. Use this option if you want to see warnings but proceed anyway. This option has no effect if the **frz** flag is ON.

**+zero\_err(# [#...])** specify message numbers that should not increment exit code

**-zero\_err(# [#...])** specify message numbers that should increment exit code

These options allow fine-grained control over exactly which messages contribute to the return value. Additionally, the **-exitcode** option allows the return value to be explicitly and unconditionally set.

When the `frz` is OFF, the return value from PC-lint is the number of messages emitted. The `+zero_err` option can be used to remove message numbers from this set and `-zero_err` can be used to add messages from this set. For example,

```
+zero_err(*)
-zero_err(w1)
-zero(80??)
```

will cause the exit code to be set to the number of error messages plus the number of messages issued in the 8000-8099 range. This option has no effect if the `frz` flag is ON.

### 4.6.3 Output

```
-env_pop pop the current option environment
-env_push push the current option environment
-env_restore(Name) restore the option environment to a previously saved one
-env_save(Name) save the current option environment with name Name
```

These options can be used to save and recall *option environments*. These options are similar to `-save` and `-restore` but operate in a larger context. Whereas `-save` and `-restore` operate only on the base suppression set state (affected by `-e#`, `+e#`, `-w#`, and `+efreeze/-efreeze`), these (`-env_*`) options also affect parameterized suppressions (`-esym`, `-efunc`, `-estring`, etc.) as well as many other options, namely: formatting options, flag options, include directory options, append options, deprecate options, library options, and reserved word options. The option environment manipulated by these options does **not** include: strong type options, message group options, return value options, function semantic options, or pre-compiled header options. The options `-skip_function`, `-unit_check`, and `-subfile` are similarly not part of the option environment and not affected by these options.

The `-env_push` and `-env_pop` options are analogous to `-save` and `-restore`. The `-env_push` option saves the current state of the option environment and a corresponding `-env_pop` option restores that state. `-env_push` options can be nested.

The `-env_save(Name)` option stores away a snapshot of the current option environment and associates it with a *Name* that can be used to later recall the option environment with `-env_restore(Name)`. The *Name* may consist of letters, numbers, and the underscore and is case-sensitive.

Attempting to use `-env_pop` without first using `-env_push` or specifying a name for `-env_restore` that was not provided as a name to `-env_save` will result in error 72 (bad option).

```
-oe(Filename) redirect stderr to Filename overwriting existing content
+oe(Filename) redirect stderr to Filename in append mode
```

This is primarily used to capture the help screen. For example:

```
lint -oe(temp) +si4 ?
```

umps the help information to file `temp` *after* the size setting has been made. If the option is introduced with a '+' as in `+oe(temp)` output is *appended* to the named file.

```
-os(Filename) redirect stdout to Filename overwriting existing content
+os(Filename) redirect stdout to Filename in append mode
```

Causes output directed to standard out to be place in the file *Filename*. This is like redirection and has the following advantages: (a) the option can be placed in a `.lint` file or anywhere that a lint option

can be placed (b) not all systems support redirection and (c) redirection can have strange side effects. If `+os` is used rather than `-os`, output is appended to the file. Make sure this option is placed before the file being linted. Thus

```
lint -os(file.out) fil.c
```

is correct. But

```
lint fil.c -os(file.out)
```

loses the intended output. The reason is that the redirection doesn't start until the option is encountered.

`-stack(&file=filename, &overhead(n), &external(n), &off, name(n))` set stack reporting options  
`+stack` enable stack reporting

The `+stack` version of this option can be used to trigger a stack usage report. The `-stack` version is used only to establish a set of options to be employed should a `+stack` option be given. To prevent surprises if a `-stack` option is given without arguments it is taken as equivalent to a `+stack` option. See [Section 17.6 Stack Usage Report](#) for more details and complete listing of the sub-options.

## 4.7 Special Detection Options

### 4.7.1 Metrics

`+metric(expr [, options] )` create, check, or nominate a metric  
`+metric_report([all | nominated])` enable metric report

See [Chapter 10 Metrics](#) for details.

### 4.7.2 Thread Analysis

`-locker_tag(type=name [, type=name...])` specify alternate locker tag class names

The standard C++ classes `std::adopt_lock_t`, `std::defer_lock_t`, and `std::try_to_lock_t` are used to specify the lock strategy when constructing a locker class object such as `std::shared_lock`. The `-locker_tag` option can be used to specify additional locker tag classes with the same semantics as one of the standard classes which may be useful when configuring thread analysis support for alternate thread libraries.

The option takes one or more arguments of the form `type=name` where `type` is one of `adopt`, `defer`, or `try_to` (specifying lock adoption, deferred locking, and try-lock semantics, respectively) and `name` is the fully qualified name of a class. The specified class will be endowed with lock strategy semantics specified by `type` when used with a function with the `mutex_tag_adopt`, `mutex_tag_defer`, or `mutex_tag_try_to_lock` semantics (see [Supporting Other Thread Libraries](#)).

`-mutex_attr([type, ] value [, mask])` specify shared/recursive values used for `pthread_mutexattr_settype`

This option is used to specify the values that indicate recursive or shared mutexes when used as an argument to a function with the `mutex_attribute_set` function semantic which also has a `mutex_is_recursive` or `mutex_is_shared` argument semantic (see [Supporting Other Thread Libraries](#)).

By default, only the `pthread_mutexattr_settype` function has the `mutex_attribute_set` function semantic with its second argument having the `mutex_is_recursive` semantic but the `-sem` option can be used to apply these semantics to other functions. PC-lint Plus will attempt to extract the values indicating a recursive mutex from the macros `PTHREAD_RECURSIVE_MUTEX_INITIALIZER` or `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`. This option can be used to specify additional values indicating a recursive mutex as well as values that indicate a shared mutex for libraries that support it.

If the first argument to `-mutex_attr` is `shared` or `recursive`, this specifies the type of mutex attribute, otherwise `recursive` is assumed. The next two arguments specify an integral *value* and *mask* with the *mask* defaulting to `-1` if not specified. The argument to a `mutex_attribute_set` function that corresponds to a `mutex_is_recursive` or `mutex_is_shared` parameter indicates a mutex attribute of *type* when the argument value bitwise anded with *mask* yields *value*.

The *value* and *mask* arguments may be specified in decimal, octal (by prefixing the number with a `0`) or hexadecimal (by prefixing the number with `0x` or `0X`). Multiple `-mutex_attr` options may be used to specify multiple value/mask combinations.

`-mutex_init(type=name [, type=name...])` specify alternate pthread mutex initialization macro names

By default, PC-lint Plus recognizes the following macros and the recursive / shared semantics implied when used to initialize `pthread_mutex_t` and `pthread_rwlock_t` objects:

- `PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP`
- `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP`
- `PTHREAD_MUTEX_INITIALIZER`
- `PTHREAD_RECURSIVE_MUTEX_INITIALIZER`
- `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`
- `PTHREAD_RWLOCK_INITIALIZER`

This option is used to specify the properties associated with additional macro names used to initialize statically allocated `pthread_mutex_t` and `pthread_rwlock_t` objects.

The option accepts one or more arguments of the form `type=name` where *name* is the name of the macro to be registered and *type* is one of `plain`, `recursive`, `shared`, or `both` indicating that the specified macro initializes a plain, recursive, shared, or shared and recursive object, respectively.

`-thread_report(type=type, file=filename [, format=format] [, filter...] [, field...])` enable a thread analysis report

The `-thread_report` option is used to trigger a thread analysis report which includes information about the use of threads, functions, variables, and/or mutexes used in a multi-threaded program. See [Thread Analysis Reports](#) for more details about the information included in the report and the available sub-options.

### 4.7.3 Strong Type

`-father(parent, child [, child...])` a stricter version of `-parent`

This option is like the `-parent()` option except that it makes the relationship a strict one such that a *child* type can be assigned to a *parent* type but not conversely. To make all relationships strict you may use the `-fhd` option. (Turn off the Hierarchy Down flag). If a `-parent()` option and a `-father()` option are both given between the same two types then the relationship is considered strict. See [Section 7.5.4 Restricting Down Assignments \(-father\)](#)

`-index(flags, ixtype, sitype [,sitype...])` establish *ixtype* as index type

This option is supplementary to and can be used in conjunction with the `-strong` option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype* (or *sitype*'s if more than one is provided). Both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a `typedef` declaration. See Section 7.3 Strong Types for Array Indices

`-parent(parent, child [,child...])` augment strong type hierarchy

This option adds a link or links to the strong type hierarchy. See Section 7.5.3 Adding to the Natural Hierarchy

`-strong(flags [,name...])` imbues typedefs with strong type checking characteristics

Identifies each *name* as a strong type with properties specified by *flags*. Presumably there is a later `typedef` defining any such *name* to be a type. Strong types are completely described in Chapter 7 Strong Types.

#### 4.7.4 Miscellaneous Detection

`-deprecate(Category, Name [,Commentary])` deprecates the use of *Name* within *Category*

The `-deprecate` option can be used to mark variables, macros, functions, keywords, types, options, and conversion specifiers as deprecated. When a deprecated entity is used in source code, message 586 will be issued warning of the use of a deprecated entity. See 17.8 Deprecation of Entities for more information.

`-idlen(count [,options])` specifies the number of meaningful characters in identifier names

This option defines the number of significant characters for different types of identifiers, as well as whether those names should be treated as case-sensitive, for the purpose of reporting name clashes in C modules. When this option is used with a value of *count* greater than zero, PC-lint Plus will report on pairs of identifiers in the same name space that are identical in their first *count* characters but otherwise different. *Options* are:

Option	Meaning
x	eXternal symbols
p	Preprocessor symbols
c	Compiler symbols

If omitted, all symbols are assumed. Uppercase versions of these options may be used to additionally specify that symbol names are case insensitive.

The C language Standards define minimum limits on the number of significant characters of an identifier. Compilers or linkers that employ such a limit will ignore all but the first *count* characters. The `-idlen` option can be used to find pairs of identifiers that are identical in the first *count* characters but are nonetheless different. PC-lint Plus treats the identifiers as different but reports on the clash.

The minimum identifier limits defined by C and the corresponding `-idlen` values are show below.



Language Version	External Identifier Minimum	Internal Identifier Minimum	Preprocessor Identifier Minimum	PC-lint Plus options
C89	6 case-insensitive characters	31 case-sensitive characters	31 case-sensitive characters	<code>-idlen(6,X)</code> <code>-idlen(31,pc)</code>
C99	31 case-sensitive characters	63 case-sensitive characters	63 case-sensitive characters	<code>-idlen(31,x)</code> <code>-idlen(63,pc)</code>
C11	31 case-sensitive characters	63 case-sensitive characters	63 case-sensitive characters	<code>-idlen(31,x)</code> <code>-idlen(63,pc)</code>

Option **x**, *external* symbols, refers to functions and variables with external linkage. Option **p**, *preprocessor* symbols, refers to macros and parameters of function-like macros. Option **c**, *compiler* symbols, refers to all the other symbols and includes symbols local to a function, struct/ union tags and member names, enum constants, etc.

Message 621 is issued when two *external* identifiers clash (across the entire program, regardless of scope), when two *compiler* identifiers or a *compiler* and an *external* identifier clash in the same name space, or when *preprocessor* identifiers clash. The description of message 621 lists all of the different cases where clashes are reported.

Message 621 may be suppressed for individual identifiers or types of clashes using the `-estring` option. `-idlen` is off (equivalent to `-idlen(0)`) by default.

```
+misra_interpret(Language*Year, interpretation) enable MISRA interpretation
-misra_interpret(Language*Year, interpretation) disable MISRA interpretation
```

These options allow for the interpretation of MISRA standards to be configured. The first argument is the language and year of a MISRA standard, one of `c2012`, `c++2008`, `c2004`, or `any`. Note that only the combinations listed below are allowed. In particular, `any` may only be used with the interpretations that specify it below and in those cases `any` must be used.

The second argument is a string which determines which interpretation is modified. These strings are descriptions of the behavior that will change. There are no shorter variants or abbreviations. Ensure you follow any necessary documentation process for your project when modifying the interpretation of MISRA rules.

The configurable interpretations are:

- **any**  
`return can replace break in switch`  
(default OFF)  
When enabled, messages 9042, 9077, and 9090 will not be issued for switch cases that are terminated by an unconditional `return` statement instead of the unconditional `break` (or `throw`) statement that is otherwise required.
- **any**  
`goto can replace break in switch`  
(default OFF)  
When enabled, messages 9042, 9077, and 9090 will not be issued for switch cases that are terminated by an unconditional `goto` statement instead of the unconditional `break` (or `throw`) statement that is otherwise required.
- **c2012**  
`essential type differs from standard type only for int and unsigned int`  
(default ON)



When enabled, the essential type of an expression will simply be the standard type of the expression unless the standard type of the expression is **(signed) int** or **unsigned int**. Otherwise, such restrictions will only apply when specifically called out by normative text. For example, the essential type of `5UL * 5UL` will be **unsigned long** when this is ON and **unsigned char** when it is OFF.

- **c2012**  
**essential type of sizeof is the UTLR of the result**  
 (default OFF)  
 When enabled, the essential type of `sizeof` and `_Alignof` expressions that yield constant values will be the unsigned type of lowest rank for the value.
- **c++2008**  
**underlying type of explicit cast is the cast type**  
 (default ON)  
 When enabled, the underlying type of an explicit cast will be the type that the subexpression was cast to even when the entire expression is an integer constant expression. Otherwise, the entire expression, including the explicit cast, will be assigned an underlying type based on the resulting constant value.
- **c++2008**  
**permit unmodified non-const incomplete array parameters**  
 (default OFF)  
 MISRA C++ requires that variables be declared using the `const` qualifier unless they are modified. MISRA C++ also requires that a pointer parameter to which array indexing is applied be declared using array syntax. It not possible to declare a `const` pointer parameter using array syntax. When enabled, this impossible combination of requirements is relaxed by permitting the lack of `const` on a parameter of pointer type that was declared as an incomplete (unsized) array. This does not affect constant size array parameters—consider using a reference to an array of constant size instead.
- **c++2008**  
**ignore constant expression underlying type rules for const variables**  
 (default OFF)  
 When enabled, the MISRA C++ rules specifying the underlying type of an integer constant expression will be ignored when determining the underlying type of an access of an integer constant defined as a `const` or `constexpr` variable. This also extends to a unary operator whose operand meets these criteria, and to a binary or ternary operator where either the left/right (or true/false) operands both meet these criteria, or one of them meets these criteria and the other is an integer constant expression.
- **c2004**  
**allow types of equal size for rule 10.1**  
 (default ON)  
 When enabled, violations of rule 10.1 will not be reported for "conversions" between identical but distinct integer types (for example, `int` and `long` could potentially be the same size). Note that even if this is disabled, the word "wider" in rule 10.1 (a) specifically will be interpreted as "not smaller" as required for the rule's own examples.

#### 4.7.5 Semantic

`-function(Function0 [,Function1] [,Function2...])` copy or remove semantics from *Function0*

This option specifies that *Function1*, *Function2...* are like *Function0* in that they exhibit special properties normally associated with *Function0*. The special functions with built-in meaning are

described in Section 9.1 [Function Mimicry \(-function\)](#). See also `-sem` in Section 9.2 [Semantics Specifications](#).

`-printf(N, name1 [,name2...])` specified *names* are `printf`-like functions with format provided in the *N*th argument

This option specifies that *name1*, *name2*, etc. are functions that take `printf`-like formats. The format is provided in the *N*th argument. For example, `lint` is preconfigured as if the following options were given:

```
-printf( 1, printf )
-printf( 2, sprintf, fprintf )
```

For such functions, the types and sizes of arguments starting with the variant portion of the argument list are expected to agree in size and type specified by the format. The variant portion of the argument list begins where the ellipsis is given in the function declaration. For non-variadic functions, the first data argument is expected to appear after the format argument. See also `-scanf` below and Sections 9.1 [Function Mimicry \(-function\)](#) and 17.1 [Format Checking](#).

`-scanf(N, name1 [,name2...])` specified *names* are `scanf`-like functions with format provided in the *N*th argument

This option specifies that *name1*, *name2*, etc. are functions that take `scanf`-like formats. The format is provided in the *N*th argument. For example, `lint` is preconfigured as if the following options were given:

```
-scanf( 1, scanf )
-scanf( 2, sscanf, fscanf )
```

For such functions, the types and sizes of arguments following the *N*th argument are expected to be pointers to arguments that agree in size and type with the format specification. See also `-printf` above.

`-sem(Function [,Sem...])` associates the semantic *Sem* with *Function*

This option allows the user to endow his functions with user-defined semantics, or modify the pre-defined semantics of built-in functions. For example, the library function `memcpy(a1,a2,n)` is pre-defined to have the following semantic checking. The third argument is checked to see that it does not exceed the size (in bytes) of the first or second argument. Also, the first and second arguments are checked to make sure they are not NULL.

To represent this semantic you could have used the option:

```
-sem( memcpy, 1P >= 3n && 2P >= 3n, 1p, 2p)
```

The details of semantic specifications are contained in Section 9.2 [Semantic Specifications](#).

#### 4.7.6 Value Tracking

`-specific_climit` maximum number of specific calls per function

The total number of specific calls recorded for any one function is limited to *n*. Because of recursion, the total number of specific calls made on any one function can be huge. This option prevents any one function from hogging resources. By default, the value is 0, implying no limit.

`-vt_depth=n` specifies the maximum number of nested specific walks

The maximum call stack depth for specific walks during value tracking. This limits the number of nested specific walks and also acts as an upward limit on recursion depth.

`-vt_passes=n` specifies the number of passes for intermodule value tracking

The number of times the entire set of project modules will be rescanned to perform additional intermodule value tracking using specific calls saved from the previous pass. Each module is processed during the initial pass where intramodule messages are issued and again during global wrap-up (if it has not been disabled), and each of these constitute a pass. This option can be used to request additional auxiliary passes beyond those that arise naturally from the processing architecture.

## 4.8 Meta Characters for Options

The following meta characters may be used within names that are used as arguments for options such as: `-esym`, `-efile`, `-emacro`, `-efunc`, `-estring`, `-etype`, `-ecall`, `-libdir`, `-libh`, `-libm` and the versions of these options that begin with `+`.

- \* wild card character matching 0 or more characters
- ? wild card character matching any single character
- ` backtick used to escape any meta character
- [...] bracketed string meaning optional matches
- "..." used when incorporating comma (,) or unbalanced '(' or ')’ with an argument

Although an option containing meta characters may not always be placed on a command line because the special characters may trip up the shell (command interpreter), it can be placed in a `.lnt` file.

Arguments (both error numbers and symbols) may contain 'Wild-card' characters.

For example

```
-esym( 715, un_* )
```

suppresses message 715 for any symbol whose first three characters are "un\_". As another example:

```
-esym( 512, ?, *::* )
```

suppresses message 512 for any symbol name containing exactly one character and for any name containing a ":", i.e. for any member name. As another example:

```
-esym( *, name )
```

suppresses any message about symbol `name`.

A string of the form [...] means that the string bounded by square brackets is optional. E.g.

```
-esym( 768, [A::]alpha )
```

will suppress message 768 for symbols "alpha" and "A::alpha". Wild-card characters may appear within the brackets. Thus

```
-esym( 768, [*::]alpha )
```

will suppress 768 for any symbol where name is "alpha" no matter how deeply nested within classes and or namespaces it may be. The accent grave character ` is sometimes referred to as the backtick. It can be used to escape any of the meta characters. For example

```
-esym( 1533, operator* )
```

suppresses Warning 1533 for any function whose name begins with "operator". However

```
-esym( 1533, operator`* )
```

suppresses 1533 for the function named `operator*`.

Within options, commas separate arguments. But suppose your argument contains a comma. For this you may use the double-quote. Example,

```
//lint -etype(1502, "B<float, int>")
// Without the double quotes, -esym() sees three arguments:
// "1502", "B<float", and "int>"
template <class T, class U> class B { };
B<float, int> b;
```

Both wildcard characters and non-wildcard characters may be used both within and outside of the double-quoted sequence. A right parenthesis or comma that appears outside of a double-quoted sequence marks the end of the argument as usual.

The character literals `"`, ```, `*`, `?`, `[`, and `]` may be expressed by escaping them with a backtick:

```
`"   ``  `*   `?   `[   `]
```

As a special case, the string `"[]"` (as in `"operator[]"` or `"extern int a[];"`) need not be escaped since, as a wildcard pattern, it is meaningless.

All escape sequences other than those mentioned above are reserved for future use. Currently, when we encounter such a sequence, we will ignore the backtick and issue a warning. For example, ``a` is taken as `"a"`.

As a special exception, a pattern that consists entirely of `?` and/or `*` characters is treated as if the pattern were replaced by `*`. This means, for example, that `-e?`, `-e??`, `-e???`, and `-e????` are all equivalent to `-e*` which disables all messages, i.e. `-e?` does not disable only messages in the range 1-9.

The patterns supplied to `+libdir/-libdir`, `+libh/-libh`, `+libm/-libm`, and `-efile/+efile` are matched against files or directories and the following additional properties apply to those patterns:

1. If the `fff` flag is enabled, the match will be performed in a case-insensitive manner.
2. Forward slashes and backward slashes are interchangeable, e.g. a backward slash in a the pattern will match a forward slash in the file or directory being matched.

## 4.9 How Suppression Options are Applied

PC-lint Plus has a rich set of options that govern when and how diagnostics are issued. It is possible for multiple options to affect the issuance of a particular diagnostic. This section describes the process by which PC-lint makes the determination to issue a specific diagnostic, including the interaction of multiple relevant suppression options.

Note that the term "suppression options" is used here to refer both to options that suppress a message (e.g. `-e#`) and options that enable a message (e.g. `+esym`).

PC-lint Plus performs the following steps, in the order given, to determine if the message should be suppressed or issued.

1. If the message is an unsuppressible message (errors 305, 309, 315, 330, and 367), the message is issued.

2. If a single-line suppression exists on the same line as the location given in the message, and the message is not frozen (with the `+efreeze` or `++efreeze` options) at the point of the suppression, the message is suppressed.
3. If the location of the message is subject to a scoped suppression (`-e(#)`, `--e(#)`, `-e#`, or `--e#`, including those resulting from `-emacro((#))`, etc. options), and the message is not frozen at the point of suppression, the message is suppressed.
4. A voting mechanism is next employed to determine whether to issue the message. If the message is currently in the set of enabled messages (this is the message set that is manipulated by the `-w/-e/+e` options), one vote is cast in favor of issuing the message. Each `-esym`, `-etype`, `-estring`, `-equery`, `-egrep`, `-efile`, `-ecall`, `-efunc`, `-elibcall`, `-elibsymb†`, `-elibmacro‡`, and `-emacro` option that applies to the current message casts its votes<sup>†</sup> in favor of suppressing the message for each parameter in the message that is matched by the option. Similarly, each applicable `+esym`, `+etype`, `+estring`, `+egrep`, `+equery`, `+efile`, `+emacro`, `+ecall`, `+efunc`, and `+elibcall` option casts its votes in favor of emitting the message. If the number of votes to suppress is greater than or equal to the number of votes to issue the message, the message is suppressed.
5. If the message location refers to a library region, and the message is suppressed for libraries via the `-elib` or `-wlib` options, the message is suppressed<sup>‡</sup>.
6. The message is issued.

<sup>†</sup> Parameterized suppression options cast one vote for each parameter in the message that matches the provided pattern.

<sup>‡</sup> Global wrap-up messages targeting symbols that were ever declared in a non-library region are treated as non-library regardless of the location used to issue the message unless the `f1o` flag is active. Library suppressions are not applied to such messages.

### Example

Elective Note 9001 (octal constant `'String'` used) is parameterized by a single string (the octal constant encountered). Given a file `example.c` that contains:

```
int i1 = 00;
int i2 = 01;
```

If PC-lint Plus is run with the options `-w1 +e9001` (disabling all non-error messages and then enabling note 9001), the messages issued are:

```
note 9001: octal constant '00' used
note 9001: octal constant '01' used
```

The `+e9001` results in one vote in favor of issuing the messages and nothing casts a vote against. If we add the option `-efile(9001, example.c)`, both messages will be suppressed because for each message the `+e9001` effects one vote in favor of issuing and the `-efile` option votes against issuing the note. Since there are not more votes to issue than to suppress, the message is suppressed. If we also add the option `+estring(9001, 00)` an extra vote will be cast to issue the message when the message references the octal constant 00 resulting in the output:

```
note 9001: octal constant '00' used
```

In other words, when using the options `+e9001 -efile(9001, example.c) +estring(9001, 00)`, message 9001 will be allowed in all files except `example.c` where only uses of the octal constant 00 will be reported.

## 4.10 Rules for Parameterized Suppressions

The parameterized suppression options are:

```
-esym, -etype, -estring, -equery, -egrep, -efile, -ecall, -efunc, -emacro,
+esym, +etype, +estring, +equery, +egrep, +efile, +ecall, +efunc, and +emacro.
```

A parameterized suppression option is ignored if an identical option is already in effect in the current option environment. For example, the options `-esym(714,foo)` `-esym(714,foo)` cannot be used to cause two votes against message 714 when issued for symbol `foo` as the second identical option will be ignored.

Two parameterized suppression options of the same kind are considered to be identical if they specify equivalent *effective message sets* and identical *parameter patterns*. The *effective message set* for parameterized message *disabling* options (those starting with "-") is the specified message set minus any messages that are currently frozen via `+efreeze`/`++efreeze` options. For parameterized message *enabling* options (those starting with "+"), the *effective message set* is always the same as the specified message set. Two *parameter patterns* are considered identical only if their spelling is identical.

This means, for example, that `-esym(71?,foo)` is considered identical to `-esym(710 711 712 713 714 715 716 717 718 719,"foo")` (assuming that no messages in the range 710-719 are frozen) as both options have equivalent *effective message sets* (710-719) and identical *parameter patterns* (`foo`, argument delimiting quotes are not considered to be part of the spelling).

If message 714 was frozen when the first `-esym(71?,foo)` option was seen, the same option appearing when 714 was no longer frozen would be *not* be ignored because the *effective message sets* of the two options are different.

The option `-esym(714,"[f]oo")` is distinct from `-esym(714,"foo")` because the *parameter patterns* are not identical and both options will cast a vote against issuance of message 714 for a symbol `foo`.

A "-" version and a "+" version of a parameterized message suppression option with identical arguments may exist at the same time with each being able to cast a vote at message issuance time. This distinction is relevant in the following example:

```
//lint -esym(714,foo)
... // region A
//lint +esym(714,foo)

//lint -esym(714,foo)
... // region B
//lint +esym(714,foo)
```

The intent was to suppress message 714 for symbol `foo` only within regions A and B. The effect is obtained for region A since the first `-esym` option will cast a vote against the message and the first `+esym` will effectively negate the first `-esym` by casting a vote for the message. Since both options remain in effect and, as previously stated, later identical parameterized suppressions are ignored, the second `-esym` option will have no effect and message 714 will not be suppressed within region B (the first `-esym` will cast one vote against issuance which will be canceled out by the first `+esym` casting a vote for issuance). The desired effect can be obtained through the use of temporary option environments using `-env_push` and `-env_pop`:

```
//lint -env_push -esym(714,foo)
... // region A
//lint -env_pop

//lint -env_push -esym(714,foo)
... // region B
//lint -env_pop
```

## 4.11 Flag Options

Options beginning with `+f`, `++f`, `-f`, or `--f` introduce flags. A flag is represented internally by an integer and is considered

ON if the integer > 0  
 OFF if the integer <= 0

Default settings are either 1 if ON or 0 if OFF.

**+f...** turns a flag ON by setting the flag to 1.  
**-f...** turns a flag OFF by setting the flag to 0.  
**++f...** increments the flag by 1.  
**--f...** decrements the flag by 1.

The latter two operations are useful in cases where you may want to turn a flag ON locally without disturbing its global setting. For example:

```
/*lint ++flb */
int printf();
/*lint --flb */
```

can be used to set the `flb` (library) flag ON for just the one declaration and, afterward, restoring the value of the flag to whatever it had been.

The table below summarizes the available flag options. Flags that were introduced in PC-lint 9 or earlier are marked — in the version column.

Flag	Default	Summary	Version
<a href="#">f12</a>	OFF	view MISRA C 2012 essential types	1.0
<a href="#">f@m</a>	OFF	commerical '@' is a modifier	—
<a href="#">faa</a>	OFF	enable aligned allocations	2.1
<a href="#">fac</a>	OFF	allow instantiation of abstract classes	1.0
<a href="#">faf</a>	OFF	enable tracking of all function call locations for thread reports	1.4
<a href="#">fai</a>	ON	arguments pointed to get initialized	—
<a href="#">fan</a>	OFF	support anonymous unions	—
<a href="#">fas</a>	OFF	support anonymous structs	—
<a href="#">fat</a>	ON	parse .net attributes	—
<a href="#">fau</a>	OFF	bitwise AND with negative constant is unknown	1.2
<a href="#">fav</a>	OFF	enable tracking of all variable access locations for thread reports	1.4
<a href="#">fba</a>	OFF	bit addressability	—
<a href="#">fbe</a>	OFF	enable backslash escapes for special option characters	1.1
<a href="#">fbl</a>	OFF	dependent base class lookup in templates	1.0
<a href="#">fbo</a>	ON	activate <code>bool</code> , <code>true</code> , <code>false</code>	—
<a href="#">fca</a>	ON	convert attributes to semantics	1.0
<a href="#">fcc</a>	OFF	capitalize message categories	1.0
<a href="#">fce</a>	OFF	continue on <code>#error</code>	—
<a href="#">fcm</a>	ON	copy semantics from macro definitions	1.0
<a href="#">fcn</a>	ON	convert non-printable characters in context line	1.0
<a href="#">fcp</a>	OFF	all subsequent modules are considered C++	—
<a href="#">fcs</a>	OFF	continue on static assertion failure	1.0
<a href="#">fcu</a>	OFF	<code>char</code> is unsigned	—
<a href="#">fcv</a>	ON	don't report parameter could be const if exclusively cast to void	1.3
<a href="#">fcw</a>	ON	attribute responsibility for last write in callee to caller	1.2
<a href="#">fdd</a>	ON	dimensional by default	—
<a href="#">fde</a>	OFF	initialization with explicitly declared default constructor is explicit	1.3.5

Flag	Default	Summary	Version
<a href="#">fdg</a>	ON	expansion of digraphs	—
<a href="#">fdh</a>	OFF	append '.h' to header names in <code>#include</code> 's	—
<a href="#">fdi</a>	ON	search directory of including file	—
<a href="#">fdl</a>	OFF	pointer difference is <code>long</code>	—
<a href="#">fdm</a>	OFF	comma from macro expansion does not delimit macro args	1.0
<a href="#">fdt</a>	OFF	delayed template parsing	1.0
<a href="#">fdx</a>	OFF	consider use of operator delete to be a modification	1.0
<a href="#">fee</a>	ON	expand environment variables	1.0
<a href="#">fei</a>	OFF	underlying type for <code>enum</code> is always <code>int</code>	—
<a href="#">fes</a>	OFF	search enclosing scopes for friend tag decls	1.0
<a href="#">fet</a>	OFF	require explicit throw specifications	—
<a href="#">ffb</a>	ON	for loop creates separate block	—
<a href="#">ffc</a>	ON	non-library functions assume custody of non-const pointers	—
<a href="#">fff</a>	OFF	fold filenames to a consistent case	—
<a href="#">ffi</a>	OFF	format integers relative to nearby limits	1.3
<a href="#">ffj</a>	ON	join within one-way hierarchy yields base	1.4
<a href="#">ffn</a>	OFF	use full file names	—
<a href="#">ffv</a>	OFF	implicit function to void pointer conversion	1.0
<a href="#">ffw</a>	OFF	allow friend decl to act as forward decl	1.0
<a href="#">fgi</a>	OFF	<code>inline</code> treated as GNU inline	1.0
<a href="#">fgl</a>	ON	use GNU line markers in preprocessed output	1.2
<a href="#">fhd</a>	ON	allow hierarchy downcasts	—
<a href="#">fho</a>	OFF	header include guard optimization	1.0
<a href="#">fhs</a>	ON	natural hierarchy of strong types	—
<a href="#">fhx</a>	ON	hierarchy of index types	—
<a href="#">fia</a>	OFF	inhibit supplementary messages	1.0
<a href="#">fie</a>	OFF	use the integer model for enums	—
<a href="#">fim</a>	ON	<code>-i</code> can have multiple directories	—
<a href="#">fin</a>	OFF	refer to supplemental messages with the info label	1.0
<a href="#">fit</a>	OFF	thrown exception renders function not declared noexcept impure	2.0
<a href="#">fiw</a>	ON	initialization is a write	—
<a href="#">fiz</a>	ON	initialization by zero is a write	—
<a href="#">fkp</a>	OFF	use K&R preprocessor	—
<a href="#">fla</a>	ON	locations for all diagnostics	1.0
<a href="#">flb</a>	OFF	treat code as library	—
<a href="#">flf</a>	OFF	process library functions	—
<a href="#">fll</a>	OFF	allow <code>long long int</code>	—
<a href="#">flm</a>	OFF	lock message format	—
<a href="#">fln</a>	ON	honor <code>#line</code> directives for diagnostics	—
<a href="#">flo</a>	OFF	library declarations override non-library declarations	1.4
<a href="#">flp</a>	OFF	lax null pointer constants	1.0
<a href="#">fls</a>	OFF	assume external call chains modify static local variables	1.3.5
<a href="#">fma</a>	OFF	microsoft inline <code>asm</code> blocks	1.0
<a href="#">fme</a>	ON	enable metrics	2.0
<a href="#">fmi</a>	OFF	enable supplemental mutex information messages	1.4
<a href="#">fml</a>	OFF	metrics for library entities	2.0
<a href="#">fms</a>	OFF	microsoft semantics	1.0
<a href="#">fmt</a>	OFF	match template template-arguments to compatible templates	1.2



Flag	Default	Summary	Version
<a href="#">fmx</a>	ON	enable member access control in C++	1.2
<a href="#">fnc</a>	OFF	nested comments	—
<a href="#">fnf</a>	OFF	fall back to operator <b>new</b> when <b>new[]</b> not available	1.0
<a href="#">fnn</a>	OFF	new can return null	—
<a href="#">fnr</a>	OFF	null pointer return	—
<a href="#">fnz</a>	ON	null pointers correspond to the integral value zero	1.3.5
<a href="#">fon</a>	ON	support for C++ operator name keywords	—
<a href="#">fpa</a>	OFF	pause before exiting	—
<a href="#">fpe</a>	ON	use precision of enumerators instead of explicit enum base type	1.2
<a href="#">fpm</a>	OFF	limit precision to the maximum of the arguments	—
<a href="#">fpu</a>	OFF	pointer parameter may be null	—
<a href="#">fpo</a>	ON	limit precision to the type of the operation	—
<a href="#">fqb</a>	ON	qualifiers go before types	—
<a href="#">fql</a>	ON	execute queries in library regions	2.1
<a href="#">frc</a>	OFF	remove commas before <code>__VA_ARGS__</code>	1.0
<a href="#">frd</a>	OFF	redefine default params for class template function members	1.0
<a href="#">frz</a>	ON	use return code only to indicate execution failure	1.0
<a href="#">fsc</a>	OFF	strings are <code>const char*</code> even in C	—
<a href="#">fsd</a>	OFF	output stack diagrams	1.0
<a href="#">fse</a>	OFF	use smallest underlying type for enums	1.0
<a href="#">fsf</a>	OFF	display function names for semantics during calls	1.0
<a href="#">fsi</a>	OFF	search <code>#include</code> stack	1.0
<a href="#">fsl</a>	OFF	single line comments	1.0
<a href="#">fsn</a>	ON	treat strings as names	—
<a href="#">fso</a>	OFF	return semantics override deduced return values	1.0
<a href="#">fsp</a>	ON	specific calls	—
<a href="#">fsv</a>	ON	track static variables	—
<a href="#">fta</a>	ON	enable typographical ambiguity checks	—
<a href="#">ftc</a>	ON	enable thread analysis checks	1.4
<a href="#">ftg</a>	ON	permit trigraphs	—
<a href="#">fub</a>	ON	ignore unreachable break in switch	1.3
<a href="#">fum</a>	OFF	user declared move deletes only corresponding copy	1.0
<a href="#">fun</a>	OFF	issue additional stack usage notes	1.0
<a href="#">fup</a>	OFF	treat null pointer values as unknown after reporting them	1.3
<a href="#">fur</a>	OFF	allow unions to contain reference members	1.0
<a href="#">fuu</a>	OFF	treat uninitialized values as unknown after reporting them	1.3
<a href="#">fvd</a>	OFF	interactive value tracking debugger	1.0
<a href="#">fwu</a>	OFF	<code>wchar_t</code> is unsigned	—
<a href="#">fxt</a>	OFF	extern C functions can throw exceptions	1.3
<a href="#">fzd</a>	OFF	enable sized deallocations	1.2
<a href="#">fzl</a>	OFF	<code>sizeof</code> is <code>long</code>	—
<a href="#">fzu</a>	ON	<code>sizeof</code> is unsigned	—

**f12** view MISRA C 2012 essential types (default OFF).

Issues message [9903](#) after full expressions to illustrate the MISRA C 2012 essential types involved in the expression and how they combine to form new essential types.

For example, given:

```
int f(int a, short b) {
    return (b + b) + (a + b);
}
```

PC-lint Plus emits:

```
note 9032: left operand to + is a composite expression of type
'signed16' which is smaller than the right operand of type 'signed32'
return(b + b) + (a + b);
~~~~~ ^
```

Why is the left side `signed16` and the right side `signed32`?

Processing the example with `+f12` and `+e9903` yields the step by step evaluation of the expression with the corresponding essential types involved at each step:

```
info 9903: (signed16 + signed16) + (signed32 + signed16)
info 9903: (signed16) + (signed32)
info 9903: signed32
```

**f@m** commercial '@' is a modifier (default OFF).

This is a feature required by some embedded compilers that employ a syntax such as:

```
int @interrupt f() { ... }
```

the `@interrupt` serves as a modifier for the function `f` (to indicate that `f` is an interrupt handler). Normally '@' would not be allowed as a modifier. If the option `+f@m` is given then '@' can be used in the same contexts as other modifiers. There will be a warning message (430) but this can be suppressed with a `-e430`. The '@' will otherwise be ignored. The keyword that follows should be identified either as a macro with null value as in `-dinterrupt=` or as a reserved word using `+rw(interrupt)`.

**faa** enable aligned allocations (default OFF).

This flag controls whether the C++17 aligned allocation feature is enabled.

Enabling this flag will allow for the recognition of operator new and delete global replacement functions. This may solve some parse errors related to aligned allocations when the feature is in use.

**fac** allow instantiation of abstract classes (default OFF).

Some compilers allow instantiation of abstract classes (e.g. Visual C++ 6). If this flag is ON, such instantiations will be allowed, otherwise an error will be emitted and the class will not be instantiated.

**faf** enable tracking of all function call locations for thread reports (default OFF).

If this flag is ON, all function call locations will be emitted in the thread analysis function report (see see [Function Reports](#)). When this flag is OFF, only the first location of each called function with a unique mutex lock ordering is reported. Setting this flag enables collection of the additional information required to report all calls.

**fai** arguments pointed to get initialized (default ON).

When an argument is passed to a function in the form of a pointer or reference to an uninitialized object and the receiving parameter is a pointer or reference to non-`const`, it is assumed by default that the object will be initialized to an unknown value. Thus, in the following:

```
void f(int*);
int g() {
    int x;
    f(&x);
    return x; // OK, no warning.
}
```

we do not warn of the possibility that `x` is uninitialized due to the presumed initialization afforded by the function `f`. However, if the flag is turned OFF (using `-fai`), then we will warn when `x` is read in the `return` statement. If the function `f` is intended to accept and modify a previously initialized value, then the `inout` semantic can be applied to the parameter. Use of an uninitialized value would then be reported only at the call to `f` regardless of the flag state. See Section [9.2.1 Possible Semantics](#).

**fan** support anonymous unions (default OFF).

If this flag is ON, anonymous unions are supported in C90 and C99 modes (they were added to the language in C11). Anonymous unions appear within structures and have no name within the structure so that they must be accessed using an abbreviated notation. For example:

```
struct abc {
    int n;
    union { int ui; float uf; };
} s;
```

In this way a reference to one of the union members (`s.ui` or `s.uf`) is made as simply as a reference to a member of the structure (`s.n`).

**fas** support anonymous structs (default OFF).

If this flag is ON, anonymous structs are supported in C90 and C99 modes (they were added to the language in C11) as well as in C++. Anonymous structs are similar to anonymous unions. That is, if a struct has no tag and no declarator as in:

```
struct X {
    struct { int a; int b; };
} x;
```

then references to the members of the inner struct are as if they are members of the containing struct. Thus `x.a` refers to member `a` within the unnamed struct within `struct X`.

**fat** parse `.net` attributes (default ON).

Dot net (`.net`) attributes are contained within square brackets. E.g.

```
[propget, id(1)] void f( [out] int *p );
```

The square brackets and information contained therein are non standard extensions to the C/C++ standards supported by the Microsoft Visual C 7.00 and later compilers. Remarkably this doesn't

appear to interfere (or be ambiguous) with other uses of square brackets within the language. For this reason the flag is normally ON. To turn off such processing use `-fat`

**fau** bitwise AND with negative constant is unknown (default OFF).

If this flag is ON, Value Tracking will treat the result of a bitwise AND operation between an unknown value and constant value as unknown. If it is OFF, the result will be based on the value of the constant and the range of the type of the unknown operand.

**fav** enable tracking of all variable access locations for thread reports (default OFF).

If this flag is ON, all accesses to each variable will be emitted in the thread analysis function report (see see [Function Reports](#)). When this flag is OFF, only the first location of each variable access with a unique access type and set of owned mutexes is reported. Setting this flag enables collection of the additional information required to report all accesses.

**fba** bit addressability (default OFF).

If this flag is ON (by default it is OFF), an individual bit of an int (or any integral) can be specified using the notation:

*a.integer-constant*

where *a* is an expression representing the integral and *integer-constant* is an integer constant. The construct is treated as a bit field of length 1.

For example, the following code:

```
/*lint +fba Turn on Bit Addressability */
int n;
n.2 = 1;
n.4 = 0;
...
if( n.2 || n.4 ) ...
```

will set the 2nd bit of `n` to 1 and the 4th bit to 0. Later it tests those bits. This syntax is not standard C/C++ but does represent a common convention used by compilers for embedded systems.

**fbe** enable backslash escapes for special option characters (default OFF).

When the **fbe** flag is enabled, the backslash character can be placed immediately before any of the following special characters to remove the special meaning of that character:

`{ } ( ) [ ] ! , " \`

When this flag is enabled, any other character following a backslash inside an option will be met with error 72 (bad option). The **fbe** flag can be turned ON and OFF between options.

**fb1** dependent base class lookup in templates (default OFF).

When this flag is ON, unqualified lookup in a class template will result in a search of dependent base classes. This is non-standard behavior implemented by some compilers.

**fbo** activate `bool`, `true`, `false` (default ON).

If this flag is ON, keywords `bool`, `true`, and `false` are activated at the start of every C++ module.

**fca** convert attributes to semantics (default ON).

If this flag is ON, certain attributes appearing in function declarations using the GCC attribute syntax will automatically be converted to function semantics. The supported attribute to semantic mappings are:

Attributer	Equivalent
<code>format(printf, index, first-to-check)</code>	<code>-printf(index, func)</code>
<code>format(scanf, index, first-to-check)</code>	<code>-scanf(index, func)</code>
<code>noreturn</code>	<code>-sem(func, r_no)</code>
<code>nonnull(i)</code>	<code>-sem(func, iP)</code>
<code>const</code>	<code>-sem(func, pure)</code>
<code>pure</code>	<code>-sem(func, pure)</code>

For example, given the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
    __attribute__((format (printf, 2, 3)));
```

the effect of the option `-printf(2, my_printf)` is automatically applied to the function. Semantics applied from attributes only affect the overload in which the attribute appears.

**fcc** capitalize message categories (default OFF).

In PC-lint, the message categories (Error, Warning, Info and Note) were spelled with an initial uppercase letter. In PC-lint Plus, these categories are presented in all lower case. Setting this flag will emulate the PC-lint 9 behavior. If the value of the flag is 2 or greater (using `++fcc`, the message categories will be presented in all upper case (ERROR, WARNING, etc).

**fce** continue on `#error` (default OFF).

PC-lint Plus will normally terminate with a fatal error (309) when a `#error` preprocessing directive is encountered. If this flag is set to a value of 2 or greater, PC-lint Plus will still emit the error message but continue processing. This may be useful to help troubleshoot incomplete configurations in a handful of circumstances but this option should never be employed in a production environment because its use will only serve to hide serious configuration issues.

When PC-lint Plus is forced to continue processing after encountering such an error, the generated AST may be incorrect/incomplete and resulting analysis can no longer be considered to be reliable. Because of this, message 686 will be issued when this option is used. Additionally, note that while processing will continue when using this option, message 309 is still emitted and cannot be suppressed. Message 309 typically indicates missing macro definitions and can usually be resolved by examining the context in which the error is emanating and defining the appropriate macro(s).

**fcm** copy semantics from macro definitions (default ON).

This flag controls when function semantics are copied from one function to another due to a macro definition. For example, if the following macro definition is encountered:

```
#define malloc xmalloc
```

calls to `malloc` would be changed to calls to `xmalloc` but the built-in function semantics for the `malloc` function would not be copied to `xmalloc`. The same issue exists for functions with user-defined function semantics.

When this flag is ON (default), semantics are copied only if the name of the new function is the same as the old function with optional underscores at the beginning and/or end of the new name and message [828](#) will be issued to announce the semantic copying. In all other cases, semantics will not be copied and warning [683](#) will be issued.

When this flag is OFF (`-fcm`), semantics are never copied and message [683](#) will always be issued when a `#define` is used to define a macro with the same name as a function that has semantics.

If this flag is set to a value of 2 (`++fcm`), semantics are always copied, producing message [828](#) each time.

Note that if the new name corresponds to a function that already has semantics associated with it, no copying is performed and no message is issued.

**fcx** convert non-printable characters in context line (default ON).

This flag controls how non-printable characters in source code lines are represented in the context line of messages. When this flag is ON, non-printable characters are presented using the syntax `<U+dddd>` where `dddd` is the hexadecimal value of the UTF32 version of the character. Note that this conversion only occurs for the context lines included in emitted messages. If the flag is OFF the character is printed as-is, without conversion.

Note: If you are using a message height of 2 (which causes the position indicator to be embedded in the context line), and your position indicator includes non-printable characters (such as ANSI/VT100 terminal escape sequences), you will need to turn this flag OFF to keep those characters from being converted.

**fcy** all subsequent modules are considered C++ (default OFF).

When this flag is ON, all subsequent modules will be processed as C++ modules, not just the ones having a distinguished extension (by default `.cpp` and `.cxx`).

**fcs** continue on static assertion failure (default OFF).

PC-lint Plus will normally terminate with a fatal error ([330](#)) when a static assertion fails. If this flag is set to a value of 2 or greater, PC-lint Plus will still emit the error message but continue processing. This may be useful to help troubleshoot incomplete configurations in a handful of circumstances but this option should never be employed in a production environment because its use will only serve to hide serious configuration issues.

When PC-lint Plus is forced to continue processing after encountering such an error, the generated AST may be incorrect/incomplete and resulting analysis can no longer be considered to be reliable. Because of this, message [686](#) will be issued when this option is used. Additionally, note that while processing will continue when using this option, message 330 is still emitted and cannot be suppressed. Message 330 typically indicates missing or incorrect scalar type sizes (which can be configured with the `-s` option), missing type definitions, or missing or incorrectly specified macros. Review the message details and the context in which the failure occurs to address the underlying cause.

**fcu** `char` is unsigned (default OFF).

If this flag is ON, plain `char` declarations are assumed to be **unsigned**. This is useful for compilers that, by default, treat `char` as **unsigned**. Note that this treatment is specifically allowed by the ANSI/ISO standard. That is, whether `char` is **unsigned** or **signed** is up to the implementation.

**fcv** don't report parameter could be const if exclusively cast to void (default ON).

When this flag is ON, PC-lint Plus will not report that a parameter could be made const (or pointer to const, etc.) if the only reference to the parameter was a cast to `void`.

**fcw** attribute responsibility for last write in callee to caller (default ON).

If this flag is ON an unread write within a called function may later be reported in the caller with message [438](#). If this flag is OFF such writes will not be visible to callers.

For example:

```

1 void f(int* p) {
2     *p = 10;
3 }
4
5 void g(void) {
6     int a;
7     f(&a);
8 }
```

will emit [438](#) when this flag is ON but not when it is OFF:

```

warning 438: last value assigned to 'a' not used
}
^
supplemental 891: previous assignment is here
    *p = 10;
    ^
```

**fdd** dimensional by default (default ON).

If this flag is ON then strong types (with the 'J' flag) will be considered equivalent to 'Jd'. The resulting behavior will be equivalent to treating the type as a physical dimension such as meters or seconds. See Section [7.4 Dimensional Analysis](#).

**fde** initialization with explicitly declared default constructor is explicit (default OFF).

When this flag is ON, PC-lint Plus will consider an implicit initialization using an explicitly-declared default constructor to constitute an explicit initialization for the purposes of messages [727](#), [728](#), and [738](#).

**fdg** expansion of digraphs (default ON).

By default, PC-lint Plus will expand digraph sequences. If this flag is turned off, digraph sequences will not be expanded.

**fdh** append '.h' to header names in `#include`'s (default OFF).

When the Dot-H flag is ON (**+fdh**) and if an extension-less header is seen, an attempt is made to open the file first with the `.h` extension and, that failing, open the file using the original name. If the flag has a value of 2 or higher, an attempt is made to open the `.h` extended name but not on the original name.

**fdi** search directory of including file (default ON).

If this flag is ON, the search for `#include` files will start with the directory of the including file (in the double quote case) rather than with the current directory. This is the standard Unix convention and is also used by the Microsoft compiler. For example:

```
#include "alpha.h"
```

begins the search for file `alpha.h` in the current directory if the **fdi** flag is OFF; or in the directory of the file that contains the `#include` statement if the **fdi** flag is ON. This normally won't make any difference unless you are linting a file in some other directory as in:

```
lint source\alpha.c
```

If `alpha.c` contains the above `#include` line and if `alpha.h` also lies in directory `source` you need to use the **+fdi** option.

**fdl** pointer difference is long (default OFF).

This flag specifies that the difference between two pointers is typed `long`. Otherwise the difference is typed `int`. This flag is automatically adjusted upon encountering a `typedef` for `ptrdiff_t`.

If the value of the flag is 2, then pointer differences are assumed to be `long long`. This can occur through the pair of options:

```
+fdl ++fdl
```

**fdm** comma from macro expansion does not delimit macro args (default OFF).

When this flag is ON, single commas from nested macro expansions are not treated as argument separators. In PC-lint 9 this behavior was implemented via the option.

```
+compiler(comma_from_macro_expansion_does_not_delimit_macro_args)
```



**fdt** delayed template parsing (default OFF).

When this flag is ON, parsing of function template definitions will occur at the end of the module instead of when the definition is initially encountered. This is necessary to properly parse certain constructs that are not technically valid C++ but are allowed (and employed) by some implementations.

**fdx** consider use of operator delete to be a modification (default OFF).

When this flag is ON, the application of `operator delete` will make its argument ineligible for the suggestion that it could be a pointer to `const`. While it is legal to delete a pointer to `const`, this can subvert the common expectation that the target of a pointer to `const` will not be changed.

**fee** expand environment variables (default ON).

This flag controls how environment variables are handled when appearing in lint options surrounded by percent signs, e.g. `%PATH%`. If this flag is OFF, environment variables are not expanded. If this flag is set to a value of 1 (the default), environment variables are expanded but not recursively. If this flag is set to a value greater than 1, environment variables are recursively expanded.

**fei** underlying type for `enum` is always `int` (default OFF).

If this flag is ON, the underlying type of enumerations will always be `'int'`. Otherwise, the Standard C/C++ rules will be used for determining the underlying type.

**fes** search enclosing scopes for friend tag decls (default OFF).

If this flag is ON, name lookup will consider enclosing scopes for unqualified friend tag declarations that are not template-ids, allowing redeclaration from an enclosing namespace. In Standard C++, only scopes within the innermost enclosing namespace are considered. Note that this lookup scope extension occurs only for types, not functions.

**fet** require explicit throw specifications (default OFF).

If the flag is OFF then the absence of an exception specification (the throw list for a function) is treated as a declaration that the function can throw any exception. This is standard C++. If the flag is ON, however, the function is assumed to throw no exception. In effect, the flag says that any exception thrown must be explicitly given. Consider

```
double sqrt( double x ) throw( overflow );
double abs( double x );
double f( double x )
{
    return sqrt( abs(x) );
}
```

In this example, `sqrt()` has an exception specification that indicates that it throws only one exception (`overflow`) and no others. The functions `abs()` and `f()`, on the other hand, have no exception specification, and are, therefore, assumed to potentially throw all exceptions. With the Explicit Throw flag OFF you will receive no warning. With the flag ON (with a `+fet`), you will receive Warning [1550](#)

that exception `overflow` is not on the throw list of function `f()`.

The advantage of turning this flag ON is that the programmer can obtain better control of his exception specifications and can keep them from propagating too far up the call stack. This style of analysis is very similar to that employed quite successfully by Java.

The disadvantage, however, is that by adding an exception specification you are saying that the function throws no exception other than those listed. If a library function throws an undeclared exception (such as `abs()` above) you will get the dreaded `unexpected()` function call. See [1, Item 14], Scott Meyers "More Effective C++".

Can you have the best of both worlds? Through the magic of macros it would appear that you can. For example, you can define a macro `Throw` as follows:

```
#ifdef _lint
    #define Throw( x ) throw(x)
#else
    #define Throw( x )
#endif
```

When linting you would turn on the `+fet` flag. You would then use the `Throw` macro for all your exception specifications that PC-lint Plus is warning you to add. These specifications will not be seen by the compiler and therefore will not get you into trouble.

Unfortunately, you will soon discover, that `Throw` doesn't handle the multiple argument case. Clearly you can define a series of separate macros, `Throw2`, `Throw3`, etc. for different argument counts. But you can also define a multiple argument macro `Throws` as follows:

```
#define Throws(X) throw X
```

Unfortunately, this requires an extra set of parentheses when you use it as in:

```
Throws((overflow,underflow))
```

But this is not necessarily a bad thing since it will alert the reader of the code that these are not seen by the compiler.

**ffb** for loop creates separate block (default ON).

The C++ standard designates that variables declared within `for` clauses are not visible outside the scope of the `for` loop. For example, in the following code, `i` cannot be used outside the `for` loop.

```
for( int i = 0; i < 10; i++ ) {
    // ...
}
// can't use i here.
```

Some compilers still adhere to an earlier practice in which variables so declared are placed in the nearest encompassing block.

By default, this flag is ON indicating that the standard is supported. If your compiler follows a prior standard you may want to turn this OFF with the option `-ffb`.

**ffc** non-library functions assume custody of non-const pointers (default ON).

This flag is normally ON. It signifies that all non-library functions will automatically assume custody of a pointer through any non-const pointer parameter. Turning this flag OFF (with a `-ffc`) will mean that a given function will not take custody of a pointer unless explicitly directed to do so via a custodial semantic for that function and argument.

```
void f( int * );           // f takes custody
void g( const int* );      // but g does not
//lint --ffc              turn the flag off
void h( int * );           // h will not take custody
//lint ++ffc              restore the flag
```

The `non_custodial` argument semantic can be used to create an exception when this flag is ON. Conversely, the `custodial` argument semantic can be used to indicate that a function takes custody when it would not be implied by this flag. See option `-sem`. See also message 429.

**fff** fold filenames to a consistent case (default OFF).

If this flag is ON, file names are processed case-insensitively such that `X.C` may refer to a file with the name `x.c`, `#include "a.h"` can be used to include a header with the name `A.H`, etc. The options `+lnt`, `+cpp`, and `+ext` are also effected, e.g. after `+cpp(cc)` both `".cc"` and `".CC"` will be considered to be C++ extensions. This flag is only intended for use with case-insensitive filesystems.

**ffi** format integers relative to nearby limits (default OFF).

When this flag is ON, Value Tracking messages portraying large integer values and ranges will be formatted relative to nearby signed or unsigned integer type limits. For example `5 + (uint32_t)-1` will be displayed as `(UINT32_MAX + 5)` and `2147480000` will be displayed as `(INT32_MAX - 3647)`. Positive values may be compared to signed or unsigned type limits regardless of the signedness of the type of the value's expression. Negative values will only be displayed relative to a signed integer type minimum. Values may match a limit exactly or be offset in either direction. The offset is limited by the magnitude of the square root of the limit (e.g. a 32-bit limit is restricted to a 16-bit offset). Integers that do not satisfy the criteria for relative formatting will still be printed literally. Each side of a range is considered independently. One, both, or neither could be displayed in relative format depending on the values.

**ffj** join within one-way hierarchy yields base (default ON).

If this flag is ON, the non-dimensional result of an additive, multiplicative, or bitwise binary operator whose operand types are within a one-way hierarchy (see `-father` and `-fhd`) will be the base type rather than the derived type. This also applies to the interpretation of compound assignment. For example:

```
//lint -w1 +e63?
//lint -strong(AJX, Number, Index)
//lint -father(Number, Index)

typedef unsigned Number;
typedef unsigned Index;

void foo(Number number, Index index) {
    Number x = number + index;
    x += index;
    Index y = number + index;
```

```

        y += number;
    }

```

The initialization and subsequent assignment to `y` will each trigger messages [632](#) and [633](#) when this flag is ON (`Number + Index → Number`) but not when it is OFF (`Number + Index → Index`).

**ffn** use full file names (default OFF).

When this flag is ON filenames reported in error messages are full path names. This can assist editors in locating the correct position within files when default directories may be different than during the linting process. If this flag is OFF (default), filenames are reported as provided to PC-lint Plus. If this flag has a negative value (`--ffn`), the filenames are reported using just the base file name.

**ffv** implicit function to void pointer conversion (default OFF).

If this flag is ON, implicit "function pointer" to "void pointer" conversions are allowed in C++ mode (they are always allowed in C mode).

**ffw** allow friend decl to act as forward decl (default OFF).

When this flag is ON set, friend declarations act as forward declarations. This is not standard C++ behavior but is supported by some compilers.

**fgi** `inline` treated as GNU inline (default OFF).

When this flag is ON, GNU inline semantics are applied to entities declared with the `inline` keyword. Namely, declarations with `inline` that are not declared `static` are externally visible even if no `extern` specifier is present. This matches the behavior of GNU90 mode.

**fgl** use GNU line markers in preprocessed output (default ON).

If this flag is ON, preprocessed output will employ GNU line marker syntax instead of the standard `#line` directives.

**fhd** allow hierarchy downcasts (default ON).

This flag is ON by default. The strong-Hierarchy-Down flag refers to assignments between strong types related to each other via the strong type hierarchy. Normally you may freely assign up and down the hierarchy without drawing a warning. With this flag set OFF, a warning will be issued whenever you assign down the hierarchy. For example:

```

typedef int X;
typedef X Y;

```

Then an assignment from an object of type `X` to a variable of type `Y` will draw a warning if the flag is turned off. See Sections [7.5.4 Restricting Down Assignments \(-father\)](#) and [7.5.3 Adding to the Natural Hierarchy](#).

**fho** header include guard optimization (default OFF).

This flag controls the handling of header files that utilize include guards. If this flag is ON, header files with valid include guards are not re-processed and as such they will not show up in verbosity messages (with **-vi** or **-va**) and lint options that appear before the include guard or in files included after the include guard will not be executed after the initial inclusion. If the flag is OFF, files will be entered every time they are referenced. Note that in neither case are the contents of the guarded region re-processed, this flag controls only whether or not the file is re-entered before making the decision to re-process. This flag does not affect headers using **#pragma once**, such files are never re-entered regardless of the value of this flag.

**fhs** natural hierarchy of strong types (default ON).

If this flag is ON (it is by default) strong types are considered to form a hierarchy based on **typedef** statements. See Section 7.5.2 [The Natural Type Hierarchy](#) and Section 7.5.3 [Adding to the Natural Hierarchy](#).

**fhx** hierarchy of index types (default ON).

If this flag is ON (it is by default) strong index types are related via the type hierarchy. See Chapter 7 [Strong Types](#). See also the **+fhs** flag.

**fia** inhibit supplementary messages (default OFF).

If this flag is ON, supplementary messages (831 and 890–899) will be suppressed.

**fie** use the integer model for enums (default OFF).

If this flag is ON, a loose model for enumerations is used (loose model means that enumerations are regarded semantically as integers). By default, a strict model is used wherein a variable of some enumerated type, if it is to be assigned a value, must be assigned a compatible enumerated value and an attempt to use an enumeration as an **int** is greeted with a (suppressible) warning 641. An important exception is an **enum** that has no tag and no variable. Thus

```
enum {false,true};
```

is assumed to define two integer constants and is always integer model.

**fim** **-i** can have multiple directories (default ON).

With this flag ON, the **-i** option may specify multiple include directories (like the INCLUDE environment variable). For example,

```
-iC:\compiler\include;C:\myinclude
```

will have the same effect as:

```
-iC:\compiler\include -iC:\myinclude
```

**fin** refer to supplemental messages with the info label (default OFF).

By default, supplemental messages are labeled as supplemental. In PC-lint, such messages were labeled as info. If this flag is ON, the PC-lint behavior is used.

**fit** thrown exception renders function not declared noexcept impure (default OFF).

If this flag is turned ON, then a throw expression within a function body will render that function impure unless the function is declared as not throwing any exceptions. This may be indicated by declaring the function **noexcept**. If a **noexcept** function contains a throw expression, the exception must be contained within the function.

When this flag is OFF, a throw expression will not render a function impure.

Note that throwing an exception is always considered to be a side effect associated with the local expression. This flag controls whether the enclosing function is further marked as impure which can affect whether a call to the function is considered to have side effects.

**fiw** initialization is a write (default ON).

This flag is normally ON. When this flag is ON, any initialization is considered a Write to the variable being initialized (unless inhibited in some other way, see the **fiz** flag below). Two successive Writes to the same variable are flagged with Info 838. Thus:

```
int n = 3;
n = 6; // Info 838
```

is normally greeted with message 838. If the flag is turned OFF (with a **-fiw**), the message would not be issued because the assignment of 6 to **n** would be considered the first Write. A subsequent Write without a Read would be unaffected by the flag and generate Info 838. See also Warning 438, Info 838 and the **-fiz** flag below.

**fiz** initialization by zero is a write (default ON).

This flag is normally ON. When this flag is ON, an initialization by 0 is considered a Write to the variable being initialized. Two successive Writes (without an intervening Read) to the same variable are flagged with Info 838. Thus in the code:

```
int n = 0;
n = 6;      // Info 838 depends on fiz flag
n = 16;     // Info 838 always
```

The assignment of 6 to **n** is normally greeted with message 838. If the flag is turned off (with a **-fiz**), the message would not be issued because the assignment to 6 would be considered the first Write. The subsequent assignment of 16 is flagged with Info 838.

See also messages 438, 838 and the **-fiw** flag above.

**fkp** use K&R preprocessor (default OFF).

ANSI/ISO C provides several facilities of the preprocessor that were not part of K&R C including the **#elif** directive and the **defined** keyword. If this flag is ON, use of these constructs will be warned about via messages 555 and 517, respectively.

**fla** locations for all diagnostics (default ON).

If this flag is ON, filename information will be provided even when the location does not correspond with a physical source file. In such cases, a representative filename that describes the location will be provided in angle brackets. For example, a location on the command line will be referenced as "**<command line>**", a location that represents a temporary buffer such as a macro expansion or string literal concatenation will be referenced as "**<scratch space>**", a location within the LINT environment variable will be referenced as "**<LINT var>**", etc. If this flag is OFF, the filename information will be empty in such cases.

**flb** treat code as library (default OFF).

If ON, code is treated as belonging to a library header (See Section [5.1 Library Header Files](#)). That is, declared objects do not have to be used or defined and messages specified by **-elib** are suppressed. This flag has been largely superseded by the notion of "Library Header Files" (See Section [5.1 Library Header Files](#)). It still has its uses though. For example, the output of PC-lint Plus in preprocess mode (i.e. using the **-p** option) will contain Lint comments bearing **++flb** before and **--flb** after the positions at which library headers were included.

This setting and unsetting of this flag is kept independently of the notion of library header (or module). Librariness of code is determined by an OR of this flag and the Librariness of the file. In this way, sections of a file can be library while the rest is not.

**flf** process library functions (default OFF).

If this flag is OFF, non-dependent library function definitions will not be fully processed. This saves some time and avoids issues stemming from library headers making use of intrinsic functions unknown to PC-lint Plus. If this flag is ON, all library function definitions will be fully processed. When set to a value of 1 (**+flf**), library functions will not be walked during value tracking analysis. To enable value tracking within library functions, set this flag to a value of 2 (**+flf ++flf**). Setting this flag to a negative value (**--flf**) will disable processing of all library function bodies, including dependent functions.

See also the **-skip\_function** option, which can be used to skip processing on a per-function basis.

**fl1** allow long long int (default OFF).

If the long-long flag is ON (option **+fl1**) then **long long int** (or just **long long**) is a permitted type, which results in an integral quantity nominally different and usually longer than a **long int**. The size of a **long long int** can be specified with the option **-sll#**. If the long-long flag is not set, then you will be warned (Info [799](#)) if an integral constant exceeds the size of a **long**.

**flm** lock message format (default OFF).

This flag can be used by GUI front ends that depend on a particular format for error messages. If this flag is ON, the Message Presentation Options (See Section [4.3.3 Message Presentation](#)) are frozen. That is, subsequent **-h**, **-width**, and the various **-format** options are ignored. Also ignored is the **-os** option. The **-os** option (See Section [4.6.3 Output](#)) designates which file will receive error messages.

**fln** honor **#line** directives for diagnostics (default ON).

By default, **#line** directives affect the location information within error messages. The option **-fln** may be used to ignore **#line** directives. See Section 18.3 **#line** and **#**.

**flo** library declarations override non-library declarations (default OFF).

Since PC-lint Plus version 1.4, a symbol is considered to be a library symbol for the purpose of issuing global wrap-up messages only if *all* of the declarations of the symbol appeared in library regions. For example, if a non-library header is included in both a library region and a non-library region, its declarations are considered to be non-library as they did not exclusively appear in a library region. Previously, such symbols were considered library if *any* of the declarations for the symbol were encountered in a library region. As most global wrap-up messages are not issued for library symbols, the previous behavior could result in unintentional message suppressions. The previous behavior can be restored by turning this flag ON. Note that the **+libh** and **+libdir** options can be used to specify headers that should always be treated as library.

Note that when issuing a global wrap-up message for a non-library symbol, the message will be treated as if it originated from a non-library region (library suppressions will not be applied) even if the location at which PC-lint Plus actually issues the message is within a library region.

**flp** lax null pointer constants (default OFF).

In C++98, a null pointer constant was defined as an integral constant expression that evaluates to zero. Post C++11 the definition was changed to "an integer literal with value zero or a prvalue of type **std::nullptr\_t**". When in C++11 and later modes, an expression such as **1 - 1** or **"\0"** will therefore not be considered to be a null pointer constant by default. If this flag is ON then the more lax C++98 semantics will be applied for such expressions.

**fls** assume external call chains modify static local variables (default OFF).

If this flag is ON, then it will be assumed that an external call that cannot be walked could modify static local variables (e.g. through a recursive call into the caller), invalidating their previous values. This flag is OFF by default, preserving the values of static local variables and precluding the possibility of external modification.

**fma** microsoft inline **asm** blocks (default OFF).

This flag enables parsing support for Microsoft ASM blocks.

**fme** enable metrics (default ON).

If this flag is OFF, then **Metrics** will be disabled.

**fmi** enable supplemental mutex information messages (default OFF).



This flag controls the supplemental message information included with certain [Thread Analysis](#) messages. By default, supplemental information is provided for relevant references of any mutex, mutex attribute, or locker object included in the primary message. If this flag is ON, all references up to the location where the primary message is issued are provided in supplemental messages. If the value of this flag is 2 or greater (using `++fmi`), supplemental messages are emitted for all mutex, mutex attribute, and locker objects up to the current location.

**fml** metrics for library entities (default OFF).

If this flag is OFF, then checks and reports associated with [Metrics](#) will skip library entities. The value of the **flo** flag determines how the final library state is determined when an entity appears in multiple different contexts.

**fms** microsoft semantics (default OFF).

Setting this flag enables a number of Microsoft specific extensions that are not of interest to other compilers and therefore do not have their own options. This flag also enables a number of undocumented features and emulates several MS-specific bugs including:

- type definition in anonymous struct or union
- pure specification on function definition defined at class scope
- **sealed**, **override**, and **\_\_except** contextual keywords
- explicit specializations within class scope
- forward references to **enum** types
- flexible array member in unions and empty classes
- support for **throw(...)** specification

**fmt** match template template-arguments to compatible templates (default OFF).

C++17 allows a template template-parameter to bind to a template argument when the template parameter is at least as specialized as the template argument but the change as currently published is incomplete which can result in ambiguity errors for previously valid code. Because of this, the feature is not enabled by default but can be enabled by setting this flag to ON.

**fmx** enable member access control in C++ (default ON).

This flag controls whether member access control is enabled in C++. If this flag is OFF, all members of classes will be accessible as if they had been declared **public**.

**fnc** nested comments (default OFF).

If this flag is ON, comments may be nested. This allows PC-lint Plus to process files in which code has been 'commented out'. Commenting out code should not be considered good practice, however. Code should be disabled by using a preprocessor conditional as it avoids the quoted star-slash problem and it automatically assigns a condition to the re-enabling of the code.

**fnf** fall back to operator **new** when **new[]** not available (default OFF).

If this flag is enabled and a placement **new[]** is called at a point where no valid array placement **new** declaration exists, instead of giving up, PC-lint Plus will try to "fall back" to a valid **operator new** function following Microsoft's behavior. For example, given:

```
void *operator new(size_t n, void *p, size_t limit);

int main() {
    char buffer[100];
    char *p = new(buffer, sizeof(buffer)) char[10];
}
```

PC-lint Plus will emit

```
error 1024: no matching function for call to 'operator new[] '
    char *p = new(buffer, sizeof(buffer)) char[10];
              ^ ~~~~~
```

since there is no valid **operator new[]** available. With the **fnf** flag enabled, the **operator new** function will be used instead and no error will be issued.

**fnn** **new** can return null (default OFF).

Turning this flag ON yields the old style **operator new**. That is, **new** may return NULL and does not throw an exception.

According to Standard C++, there are two built-in functions supporting **operator new**:

```
void *operator new( size_t ) throw( std::bad_alloc );
void *operator new[]( size_t ) throw( std::bad_alloc );
```

Rather than return NULL when there is no more allocatable space, these functions throw an exception as shown.

However, earlier versions of the language, especially before there were exceptions, returned NULL when storage was exhausted. To support this older convention, this flag was created.

When this flag is OFF, using **std::nothrow** will still be considered to possibly return null. Decrementing this flag (from the default value of 0) will force **new** to never return null, even when it is explicitly requested that **new** not throw an exception.

**fnr** null pointer return (default OFF).

This flag is normally OFF. When this flag is ON, then all functions not available to be walked that return pointers and have no other return semantic are assumed to return pointers that could possibly be NULL. For example:

```
//lint +fnr      assume unavailable functions may return null
int *f();
void g() {
    int *p = f();      // p could be NULL
    if (!p) { return; } // avoid a diagnostic
    *p = 0;            // OK now to use p
}
```

In this example, `f` was not available to be walked because no definition was present. A function may also be unavailable to be walked due to a `-skip_function` option, a `no_specific_walk` semantic, the current `-vt_depth`, or the value of the `flf` flag.

**fnz** null pointers correspond to the integral value zero (default ON).

While it is not guaranteed by the C nor C++ standards, it is commonly assumed (and often but not necessarily true) that the integral representation of a null pointer in memory is zero. When this flag is ON, Value Tracking will treat the result of converting an integral value equal to zero to a pointer as yielding a null pointer even when the integral value is not a null pointer constant. If the flag is turned off then this assumption will not be made and only null pointer constants will be recognized. For example, when this flag is ON and `-vt_depth=2` is used, this example:

```

1 void invoke_callback(void(*cb)(unsigned int), unsigned int data) {
2     cb(data);
3 }
4
5 void callback_handler(unsigned int data) {
6     char* p = (char*)data;
7     *p = 42;
8 }
9
10 int main() {
11     invoke_callback(callback_handler, 0);
12 }
```

will report:

```

7 warning 413: likely use of null pointer 'p'
   *p = 42;
   ^~
2 supplemental 894: during specific walk callback_handler(0)
   cb(data);
   ^
...
6 supplemental 831: cast from integer yields nullptr
   char* p = (char*)data;
               ^~~~~~
...
11 supplemental 831: argument passing yields 0
   invoke_callback(callback_handler, 0);
               ^
```

**fon** support for C++ operator name keywords (default ON).

When this flag is ON (the default), the C++ alternative operator names:

**and, and\_eq, bitand, bitor, compl, not, not\_eq, or, or\_eq, xor, and xor\_eq**

are recognized as keywords with meaning equivalent to the operators:

**&&, &=, \&, |, ~, !, !=, ||, |=, ^, and ^=**

respectively. If the value of this flag is 2 or higher, these operator names are also recognized in C mode (this is typically accomplished by `#including iso646.h` in C). If the flag is turned OFF, the alternative names are not recognized in C or C++.

**fpa** pause before exiting (default OFF).

When this flag is ON, PC-lint Plus will pause just before exiting (after all messages are produced), and request input through `stdin` after prompting on `stderr`. Hitting Return (i.e., Enter) should be enough to finally terminate. This option could be useful in a setup where PC-lint Plus is launched in a separate terminal window that closes when PC-lint Plus exits, before the desired output can be reviewed. This option should keep the window open. CAUTION: This option is recommended only as a trouble shooting option or as a stop gap measure. Some environments require the launched program to terminate or they themselves lock up.

**fpe** use precision of enumerators instead of explicit enum base type (default ON).

The precision of a value of `enum` type can be determined from the values representable by combinations of its enumerators or simply from the actual integer type used to represent the `enum` type. The former behavior occurs except as specified below.

If this flag is turned OFF, C++11 enumerations defined with a fixed underlying type will use the precision of the specified type instead.

If the value of this flag is negative (e.g. `-fpe --fpe`) then the actual integer type will always be used even for `enum` types without a fixed underlying type.

If the expression has `volatile`-qualified type then the actual integer type will always be used.

The actual integer underlying type is always used for enumerations that do not declare any enumerators.

**fpm** limit precision to the maximum of the arguments (default OFF).

This is used to suppress certain kinds of Loss of Precision messages (734). In particular, if multiplication or left shifting is used in an expression involving `char` (or `short` where `short` is smaller than `int`) an unwanted loss of precision message may occur. For example, if `ch` is a `char` then:

```
ch = ch * ch
```

would normally result in a Loss of Precision. This is suppressed when `+fpm` is set. This flag is automatically (and temporarily) set for operators `<<=` and `*=`. For example

```
ch <<= 1
```

is not greeted with Message 734.

**fpn** pointer parameter may be null (default OFF).

If this flag is set ON, all pointer parameters are assumed to be possibly NULL and a diagnostic will be issued if a pointer parameter is used without testing for NULL. For example:

```
void f(char *p, char *q) {
    *p = 3;      // warning only if +fpn
    q++;         // warning only if +fpn
}
```

```

void g(char *p, char *q) {
    if (p && q) {
        *p = 3;    // no warning
        q++;       // no warning
    }
}

```

For more information about this interesting test see Chapter 8 [Value Tracking](#).

**fpo** limit precision to the type of the operation (default ON).

The precision of a mathematical operation is limited by the operation itself. For left shifting and right shifting, this flag is relevant only when the right hand operand is a known value. For left shifting, the resultant precision is first presumed to be the precision of the left hand operand plus the value of the right hand operand. For right shifting, the resultant precision is first presumed to be the precision of the left hand operand minus the value of the right hand operand. In both shifting cases, if the **fpm** flag is active, the precision is reduced to the smaller of that presumed precision and the precision of the left hand operand. If the **fpo** flag is active, the (possibly reduced) precision is reduced further still to the smaller of this reduced precision and the precision of the resultant type. For example:

```

void b(int c) {
    char d;
    d = c << 28;
}

```

will result in info [734](#), loss of precision, for the assignment regardless of whether or not this flag is active. When inactive, however, the precision reported in the message for the assigned value will be 59 bits and when active the precision will be 31 bits.

Likewise,

```

void b(int c) {
    if ((c << 28) == 2147483648)
        {}
}

```

will produce [650](#), constant '2147483648' out of range for operator '==', when **fpo** is active and will not if the flag is inactive for the same reasons.

This flag also limits the precision for multiplication. Initially, the precision of the result is presumed to be the maximum precision of the operands. A tentative precision that equals the sum of the precision of the two operands is also calculated. If the higher ranked operand is signed and a sign is not possible in either of the operands, the tentative precision is reduced by one. If either operand is a power of two, the (possibly reduced) tentative precision of the result is reduced by one. If the **fpm** flag is inactive, the initial precision of the result will be discarded and replaced with this (possibly reduced) tentative precision. If the **fpo** flag is active the precision of the result will then be reduced to the smaller of the precision of the result so far calculated and the precision of the resultant type. For example:

```

void b(int c) {
    char d;
    d = c * 28;
}

```

will result in information [734](#), loss of precision, for the assignment regardless of whether or not this flag is active. When inactive, however, the precision reported in the message for the assigned value will be 36 bits and when active the precision will be 31 bits.

Likewise,

```
void b(int c) {
    if ((c << 28) == 34359738368)
        {}
}
```

will exhibit the same 650 behavior as described above in the case of left shifting for the same reasons.

**fqb** qualifiers go before types (default ON).

This flag is normally ON and in conjunction with Elective Note 963 can report on declarations in which `const` and `volatile` qualifiers do not follow a consistent pattern as to whether they appear before or after types in a type specifier. By default Note 963 will report whenever a qualifier follows a type. If the **fqb** flag is turned OFF (e.g. `-fqb`) the qualifier is expected to follow the type. For example:

```
int const x;      // by default no message
//lint +e963      turn on message 963
int const y;      // msg 963: qualifier follows type
const int z;      // no msg
//lint -fqb       reverse the message
int const a;      // no msg
const int b;      // msg 963: qualifier precedes type
```

[2] and [3] provide supporting evidence that not only is a convention useful, but that the better convention is the one rendered with `-fqb`.

**fql** execute queries in library regions (default ON).

When this flag is OFF (the default is ON), AST Queries created using the `-astquery` option are not evaluated for AST nodes appearing in library regions. The effect is similar to that which would be achieved by prefixing all AST Queries with `!getLocation.isLibraryRegion` except that 1) the flag may be set to OFF for a portion of the project to temporarily disable Query execution in library regions and 2) turning the flag OFF completely eliminates Query execution overhead in library regions which can be significant for projects that employ a large number of Queries.

The value of this flag is only considered during Query evaluation; its value at the time Queries are registered is irrelevant. When this flag is OFF, AST nodes residing in library regions are still accessible via AST traversal but AST Queries will not be evaluated for root nodes that reside in library regions. This flag does not have any effect on `-equery` options.

**frc** remove commas before `__VA_ARGS__` (default OFF).

The variadic macro feature added in C99 has a commonly encountered limitation: it requires that at least one argument be provided for the variadic portion of the argument list. For example:

```
#define LOG(format, ...) fprintf(stderr, format, __VA_ARGS__)
```

works fine when `LOG` is called with two or more arguments, e.g. `LOG("%s", "Started")`, but not when called with a single argument, e.g. `LOG("Started")`, which would expand to `fprintf(stderr, "Started",)` (note the trailing comma). As there is no facility provided by Standard C to address this limitation, various compilers have implemented their own extensions to deal with it.

GCC (and others) handle this by ascribing special meaning to the token pasting operator `##` when placed between a comma and a variable argument in a macro definition, causing the offending comma to be removed during expansion if the variable argument is left out or empty. This allows the above to be defined as:

```
#define LOG(format, ...) fprintf(stderr, format, ##__VA_ARGS__)
```

and when invoked as `LOG("Started")` will expand to `fprintf(stderr, "Started")` (no trailing comma). PC-lint Plus supports this behavior regardless of the value of this flag but will issue message [2715](#) to alert of the non-portable behavior.

MSVC removes the trailing comma even without the appearance of the token pasting operator (as in the first example). This behavior is not enabled by PC-lint Plus by default, set this flag ON to enable support for this behavior.

**frd** redefine default params for class template function members (default OFF).

When this flag is ON, default parameters for member functions of a class template may be redefined and the new value will be ignored. This behavior is implemented by MSVC.

**frz** use return code only to indicate execution failure (default ON).

When this flag is ON, the exit code of PC-lint Plus will be 0 unless there was a fatal error. If this flag is OFF, the exit code will be the number of messages emitted, up to a max of 255 and can be manipulated with the `-zero`, `-zero_err/+zero_err`, and `-exitcode` options. If this flag is ON, the options that manipulate the exit code will have no effect.

**fsc** strings are `const char*` even in C (default OFF).

When this flag is ON, string constants are considered pointers to `const char`. For example:

```
strcpy( "abc", buffer );
```

draws a diagnostic because `strcpy` is declared within `string.h` (by all the major compiler vendors) as

```
char * strcpy( char *, const char * );
```

The diagnostic is issued because a `const char *` is being passed to a `char *`.

You may think it odd that string constants are not `const char *` by default. If you set this flag ON, you will probably discover the reason. There will undoubtedly be numerous places where a function is passed a string constant where the corresponding parameter should be declared `const char *` but isn't. There will also be cases of variables that should be declared as `const char *` but aren't. Thus, you may regard this flag as a good way to ferret out places where such type checking can be tightened.

**fsd** output stack diagrams (default OFF).

When this flag is ON and stack reporting is enabled, debugging information presenting a visual aid of how stack memory was allocated within a function will be displayed. For example, for the following function:

```
void f(int x) {
    double w;
    {
```

```

        int a;
        int b;
        int c;
    }
    float f;
    {
        double r;
        double y;
    }
}

```

the `+fsd` option will produce:

```

### Auto Usage Stack Diagram for 'f' ###
x [+4] -> 4, 4
{
    w [+8] -> 12, 12
    {
        a [+4] -> 16, 16
        b [+4] -> 20, 20
        c [+4] -> 24, 24
    } [-12] -> 12, 24
    f [+4] -> 16, 24
    {
        r [+8] -> 24, 24
        y [+8] -> 32, 32
    } [-16] -> 16, 32
    } [-12] -> 4, 32
###

```

where each new allocation (or complete block) is displayed in the format:

```
name [+added_size] -> current_usage, running_maximum_usage
```

**fse** use smallest underlying type for enums (default OFF).

In C, the underlying type of enumeration is implementation-defined but must be large enough to represent all the values in the enumeration as long as all of those values can be represented in an `int`. If all the values can be represented in a smaller **signed** or **unsigned type**, that smaller type may be used. In C++, the enumeration's underlying type is the first of the following types that can represent all of the provided values: `int`, **unsigned int**, **long**, **unsigned long**, **long long**, **unsigned long long**. This flag indicates that the smallest type that can represent all of the values specified in the enumeration should be used as the underlying type.

**fsf** display function names for semantics during calls (default OFF).

When this flag is ON, every encountered function call (including constructor calls) will be accompanied by info [879](#), which provides all of the ways that the called function may be specified inside of `-sem`, `-printf`, and `-scanf` options. This is useful when attempting to provide semantics for specific function overloads or instantiations where the precise syntax may not be obvious.

If this flag is set to 2 (`+fsf ++fsf`) then [879](#) will instead be emitted for the first declaration of every function. This will include functions which are rarely called directly such as destructors.



**fsl** search `#include` stack (default OFF).

When this flag is ON, in addition to searching the current working directory and the paths specified via `-i` options, header files specified with quoted syntax (e.g. `#include "a.h"`, not `#include <a.h>`), are searched for in each of the directories of the current include stack, starting from the top.

**fsl** single line comments (default OFF).

This flag controls whether C++-style comments are available in C89 mode.

**fsn** treat strings as names (default ON).

When the flag is OFF, the `esym()` option can be used only to suppress messages parameterized by *'Symbol'*. With this flag ON, `esym()` can also be employed to suppress (or enable) messages in the same way that `-estring()` can, e.g. messages parameterized by anything other than *'Type'* (for which `-etype()` must be used). For example:

```
//lint +fsn
//lint -esym(650, <)
int f(unsigned char c)
{
    if (c < 1000) return 1; // Potential 650 ("constant out of
                          // range for operator 'String'")
    else return 0;
}
```

The `-esym()` second argument means that the 650 will not be issued when the operation (represented by the *'String'* parameter) is `<`.

**fso** return semantics override deduced return values (default OFF).

In previous versions, PC-lint would preserve the information in a user-defined return semantic even when more precise information was known about this value. The default behavior is to retain the more specific information. When the **fso** flag is ON, the PC-lint behavior is used. This applies only to functions with an implementation visible to PC-lint Plus.

**fsp** specific calls (default ON).

If this flag is ON (it is by default), Specific function call walking is supported. See [Section 8.8 Interfunction Value Tracking](#). By turning this flag OFF (using the option `-fsp`), processing can be speeded up.

**fsv** track static variables (default ON).

Controls value tracking of static variables. When this flag is ON, all static variables will be tracked. If the flag is OFF, static variables will not be tracked between modules. Decrementing the flag when it is already OFF will disable tracking of static variables even within a single module.

**fta** enable typographical ambiguity checks (default ON).

When this flag is ON, MISRA C 2012 typographical ambiguity calculations are performed. If you are not interested in these checks and do not want to incur the associated overhead of this feature, you can turn this flag off. See also message [9046](#).

**ftc** enable thread analysis checks (default ON).

If this flag is ON, thread analysis will be performed on the analyzed program (see [Thread Analysis](#)). If the value of this flag is 2 or greater (using `++ftc`), messages [2467](#) or [2468](#) will not inhibit the second phase of thread analysis (see [Inhibition of Thread Analysis](#)). Thread analysis can be disabled by turning this flag OFF (`-ftc`).

**ftg** permit trigraphs (default ON).

If this flag is ON (it is ON by default) standard C/C++ trigraphs are permitted and message [854](#) is issued when they are converted. For example `??(` is a trigraph that denotes the left square bracket (`[`). If this flag is OFF, trigraphs will not be converted and trigraph sequences will instead result in the issuance of warning [584](#).

**fub** ignore unreachable break in switch (default ON).

When this flag is ON, message [527](#) will ignore a single redundant unreachable `break` statement in a `case` or `default` labeled region of a `switch` statement to accommodate the practice of ending all `cases` with a `break` statement even when a `case` unconditionally uses `return`, `throw`, `continue`, `goto`, or a function that does not return. Further unreachable code after the first such ignored `break` statement for a particular `case` will still be reported (including additional `break` statements).

**fum** user declared move deletes only corresponding copy (default OFF).

In Standard C++, a user-declared move operation causes both the copy constructor and copy assignment function to be deleted. When this flag is set, only the corresponding operation will be deleted, e.g a user-declared move constructor will not result in the copy assignment function being deleted. This option is provided to support MSVC behavior.

**fun** issue additional stack usage notes (default OFF).

If this flag is ON, note [2901](#) (which must be separately enabled) will be issued with information about each function. This outputs the same information as note [974](#), but is not restricted to the worst-case function.

**fup** treat null pointer values as unknown after reporting them (default OFF).

When this flag is ON (`+fup`) a pointer will only be reported as null a single time until there is further reason to believe it may be null. If a potentially null pointer is dereferenced multiple times in a row, only the first will be reported. When this flag is set to 2 (`+fup ++fup`) a pointer will be completely cleared after it has been reported as null, including discarding information about what the pointer

would point to if it were not null. When this flag is set to -1 (`-fup --fup`) a pointer that has been reported as null will then be assumed not to be null on the basis that it has already been dereferenced. For example:

```

1 void f(bool b) {
2     int* a = nullptr;
3     if (b) { a = new int; }
4     *a = 0;
5     int z = *a;
6     z = 5 / *a;
7     if (a) { }
8 }
```

Analyzing this file (with the arguments `+fia -w1 +e613 +e414 +e593 +e774 -h1` for presentation) will emit:

```

4 warning 613: potential use of null pointer 'a'
5 warning 613: potential use of null pointer 'a'
6 warning 613: potential use of null pointer 'a'
6 warning 414: possible division by zero
8 warning 593: custodial pointer 'a' possibly not freed nor returned
```

If `-fup --fup` is added, then after the first instance of 613 is emitted it will be assumed that `a` cannot be null because it was dereferenced. This will avoid subsequent instances of 613 and cause message 774 to be emitted when it is tested later:

```

4 warning 613: potential use of null pointer 'a'
6 warning 414: possible division by zero
7 info 774: boolean condition for 'if' always evaluates to 'true'
  (involving variable 'a')
8 warning 593: custodial pointer 'a' possibly not freed nor returned
```

If `+fup` is added instead, the only difference will be that message 613 is only emitted once:

```

4 warning 613: potential use of null pointer 'a'
6 warning 414: possible division by zero
8 warning 593: custodial pointer 'a' possibly not freed nor returned
```

Finally, if `+fup ++fup` is used, the first 613 will completely clear the value of `a` and prevent any further messages in the example:

```

4 warning 613: potential use of null pointer 'a'
```

**fur** allow unions to contain reference members (default OFF).

C++ forbids unions to contain reference members and by default PC-lint Plus does not allow this either. If this flag is ON, reference members will be accepted inside of unions.

**fuu** treat uninitialized values as unknown after reporting them (default OFF).

When this flag is ON values will be completely cleared once they have been reported as uninitialized (and thus will not be reported as uninitialized again on a second use).

**fvd** interactive value tracking debugger (default OFF).

This flag enables the value tracking debugger. See section [8.7 Debugger](#)

**fwu** `wchar_t` is unsigned (default OFF).

This flag is used to specify the signedness of a built-in `wchar_t` type. If this flag is set ON the built-in type is of the unsigned variety, otherwise of the signed variety.

**fxt** extern C functions can throw exceptions (default OFF).

If this flag is ON, `extern "C"` functions are assumed to be capable of throwing exceptions unless specified otherwise such as by the GCC `nothrow` attribute. By default, it is assumed that `extern "C"` functions don't throw exceptions.

**fzd** enable sized deallocations (default OFF).

This option controls whether the C++14 sized deallocation feature is enabled.

**fzl** `sizeof` is long (default OFF).

If this flag is ON, `sizeof()` is assumed to be a `long` (or `unsigned long` if `-fzu` is also ON). The flag is OFF by default because `sizeof` is normally typed `int`. This flag is automatically adjusted upon encountering a `size_t` type. This flag is useful on architectures where `int` is not the same size as `long`.

If the flag has a value equal to 2, then `sizeof()` is assumed to be `long long`. Thus

```
+fzl ++fzl
```

will result in `sizeof()` being `unsigned long long x` (assuming `+fzu`).

**fzu** `sizeof` is unsigned (default ON).

If this flag is ON, `sizeof()` is assumed to return an unsigned quantity (`unsigned long` if `fzl` is also ON). This flag is automatically adjusted upon encountering a `size_t` type.

## 4.12 Compiler Adaptation

All compilers are slightly different owing largely to differences in libraries and preprocessor variables, if not to differences in the language processed. If automated configuration support is available for your compiler then the easiest way to cope with these differences is the use of the `pclp_config.py` utility provided in the distribution. See section [2.3.2](#) to get started with `pclp_config.py`. The remainder of this section describes how many compiler extensions can be supported using PC-lint Plus options.

### 4.12.1 Customization Facilities

The following are useful for supporting a number of features in a variety of compilers. With some exceptions, they are used mostly to get PC-lint Plus to ignore some nonstandard constructs accepted by some compilers.

@ Compilers for embedded systems frequently use the @ notation to specify the location of a variable.

We have for this reason support for the @ feature, which consists of ignoring expressions to its right. When we see a '@' we then give a warning (430), which you may suppress with a -e430. For example:

```
int *p @ location + 1;
```

Although warning 430 is issued, p is regarded as a validly initialized pointer to int.

\_gobble is a reserved word that needs to be activated via +rw(\_gobble).

It causes the next token to be gobbled; i.e., it and the next token are ignored. This is intended to be used with the -d option. See co-kcarm.lnt for examples.

\_ignore\_init This keyword when activated causes the initializer of a data declaration or the body of a function to be ignored.

Cross compilers for embedded systems frequently have declarations that associate addresses with variables. For example, they may have the following declarations

```
Port pa = 0xFFFF0001;
Port pb = 0xFFFF0002;
```

etc. The type Port is, of course, non-standard. The programmer may decide to define Port, for the purpose of linting, to be an unsigned char by using the following option:

```
-d"Port=unsigned char"
```

(The quotes are necessary to get a blank to be accepted as part of the definition.) However, PC-lint Plus gives a warning when it sees a small data item being initialized with such large values. The solution is to use the built-in reserved word \_ignore\_init. It must be activated using the +rw option. Then it is normally used by embedding it within a -d option. For the above example, the appropriate options would be:

```
+rw(_ignore_init)
-d"Port=_ignore_init unsigned char"
```

The keyword \_ignore\_init is treated syntactically as a storage class (though for maximum flexibility it does not have to be the ONLY storage class). Its effect is to cause PC-lint Plus to ignore, as its name suggests, any initializer of any declaration in which it is embedded.

Some compilers allow wrapping a C/C++ function prototype around assembly language in a fashion similar to the following:

```
__asm int a(int n, int m)
{ xeo 3, (n)r ; ... }
```

Note there is a special keyword that introduces such a function. This keyword may vary across compilers. To get PC-lint Plus to ignore the function body, equate this keyword with \_ignore\_init. E.g.

```
+rw(_ignore_init)
-d__asm = _ignore_init
```

\_to\_brackets is a reserved word that will cause it and the immediately following bracketed, parenthesized or braced expression, if any, to be ignored.

It needs to be activated with +rw(\_to\_brackets). It is usually accompanied with a -d option. (For example, see co-iar.lnt on the distribution media). For example, the option:

```
-dinterrupt=_to_brackets
+rw(_to_brackets)
```

will cause each of the following to be ignored.

```
interrupt(3)
interrupt[5,5]
interrupt{x,x}
```

**\_to\_eol** When **\_to\_eol** is encountered in a program (or more likely some identifier defined to be **\_to\_eol**), the identifier and all remaining information on the line is skipped.

That is, information is ignored to the End Of Line. E.g., suppose the following nonstandard construct is valid for some compiler:

```
int f( int n ) registers readonly ( 3, 4 )
{
    return n;
}
```

Then the user may use the following options so that the rest of the line following the first **')** is ignored:

```
-dregisters=_to_eol
+rw(_to_eol)
```

**\_to\_semi** is a super gobbler that will cause PC-lint Plus to ignore this and every token up to and including a semi-colon.

It needs to be enabled with **+rw(\_to\_semi)** and needs to be equated using **-d**. For example, if keyword **\_pragma** begins a semicolon-terminated clause that you want PC-lint Plus to ignore, you would need two options:

```
-d_pragma=_to_semi
+rw(_to_semi)
```

**\_up\_to\_brackets**

is a potential reserved word that will cause it and all tokens up to and including the next bracketed (or braced parenthesized) expression to be ignored. For example:

```
//lint +rw(_up_to_brackets)    activate reserved word
//lint -dasm=_up_to_brackets    asm is now an _up_to_brackets
asm ( "abc" : "def" );          // "asm" ... ')' is ignored
asm volatile ( "asm" );         // "asm" ... ')' is ignored
```

In the above we almost could have defined **asm** to be a **\_to\_brackets**. The problem is that we also needed to ignore the volatile following **asm** and so we required the use of **\_up\_to\_brackets**.

**\_\_typeof\_\_** is similar in spirit to **sizeof** except it returns the type of its expression rather than its size.

Since it is not part of standard C or C++ the reserved word must be activated with the option:

```
+rw( __typeof__ )
```

**\_\_typeof\_\_** can be useful in macros where the exact type of an argument is not known.

For example:

```
#define SWAP(a,b) { __typeof__(a) x = a; a = b; b = x; }
```

will serve to swap the values of **a** and **b**. Some compilers not only support the **\_\_typeof\_\_** facility but they write their headers in terms of it. For example,

```
typedef __typeof__(sizeof(0)) size_t;
```

assures that `size_t` will not be out of synch with the built-in type.

One such condition is output produced by the scavenger whose purpose is to extract pre-defined macro definitions from an unwitting compiler.

`-dname{definition}` is an alternative to `-dname=definition`.

`-dname{definition}` has the advantage that blanks may be embedded in the definition. Now it is true that you could use `-d"name=definition"` and so enclose blanks in that fashion but there are certain conditions, especially compiler generated macro definitions where the use of quotation marks are not suitable.

`-dname()`=*Replacement*

`-dname(identifier-list)`=*Replacement*

To induce PC-lint Plus to ignore or reinterpret a function-like sequence it is only necessary to **#define** a suitable function-like macro. However, this would require modifying source code (or use of the `-header` option) and is hence not as convenient as using this option. For example, if your compiler supports

```
char_varyingn
```

as a type and you want to get PC-lint Plus to interpret this as `char*` you can use

```
-dchar_varying()=char*
```

As another example:

```
//lint -dalpha(x,y)=((x+y)/x)
int n=alpha(2,10);
```

will initialize `n` to 6. The above `-dalpha...` option is equivalent to:

```
#define alpha(x,y) ((x+y)/x)
```

In the no parameter case, the functional expression can have any number of arguments. For example; in the following code both `asm()` expressions are ignored even though they have a different number of arguments.

```
//lint -dasm()=

void f()
{ asm("Move a,2", "Add a,b");
  asm("Jmp.x");
}
```

As with the normal (non-functional) version of the `-d` option the `+d` variant of the option sets up a macro that cannot be redefined.

#### 4.12.2 Identifier Characters

Additional identifier characters can be established. See `-$` and `-ident()` in Section [4.4.3 Tokenizing](#).

#### 4.12.3 Preprocessor Statements

See Section [18.4 Non-Standard Preprocessing](#) for special non-standard preprocessor statements. Also see the `+ppw` option.

#### 4.12.4 In-line assembly code

Compiler writers have shown no dearth of creativity in their invention of new syntax to support assembly language.

In the PC world, the most frequently used convention is (to simplify slightly):

```
asm { assembly-code }
```

or

```
asm assembly-code <new-line>
```

where `asm` is sometimes replaced with either `_asm` or `__asm`. This convention is supported automatically by enabling the `asm` keyword, using `+rw(asm)` (or `+rw(_asm)` or `+rw(__asm)` as the case may be).

But other conventions exist as well. One manufacturer uses

```
#asm
    assembly-code
#
```

For this sequence, it is necessary to enable `asm` as a pre-processor word using `+ppw(asm)`

If your compiler uses a different preprocessor word, you may use the option `+ppw_asgn`.

Another sequence is:

```
#asm
    assembly-code
#endasm
```

For this `+ppw(asm,endasm)` is needed.

Yet another convention is:

```
#pragma asm
    assembly-code
#pragma endasm
```

For this you need to define the two pragmas with

```
+pragma(asm, off)
+pragma(endasm, on)
```

#### 4.12.5 Pragmas

##### 4.12.6 Built-in pragmas

A number of compilers support the `push_macro` and the `pop_macro` pragmas.

`push_macro( name-in-quotes )`, where the *name-in-quotes* specifies a macro, is a pragma that will save the definition of the macro onto a stack. This will allow the macro to be redefined or undefined over a sub-portion of a module. Presumably this will be followed by a `pop_macro( name-in-quotes )` pragma that will restore the original macro. Thus:

```
#define N 100
#pragma push_macro( "N" )
#define N 1000
int x[N];
#pragma pop_macro( "N" )
int y[N];
```



declares **x** to be an array of 1000 integers whereas **y** becomes an array of 100 integers.

Note that you have, in effect, **k** different stacks where **k** is the number of different names provided as arguments to these pragmas.

#### 4.12.7 User pragmas

**+pragma( *identifier*, *Action* )** associates *Action* with *identifier* for **#pragma**  
**-pragma( *identifier* )** disables pragma *identifier*

The **+pragma( *identifier*, *Action* )** option can be used to specify an *identifier* that will be used to trigger an *Action* when the *identifier* appears as the first identifier of a **#pragma** statement. *Action* must be one of

```
on
off
once
message
ppw
macro
fmacro
options
include_alias
```

Please note that the purpose of the **+pragma** option is compatibility with your compiler. If your goal is to conditionally compile depending on the presence of PC-lint Plus, use the **\_lint** preprocessor variable.

**on** and **off** – An **off** action will turn processing off. An **on** option will reset processing. For example, assume that the following coding sequence appears in a user program:

```
#pragma ASM
    movereg 3,8(4)
#pragma ENDASM
```

Lint will normally ignore the **#pragma** statements but it will not ignore the assembly language between the **#pragma** statements, which might lead to a flurry of messages. To resolve the problem, add the pair of options:

```
+pragma( ASM, off )
+pragma( ENDASM, on )
```

This will turn off lint processing when the **ASM** is seen and turn it back on when the **ENDASM** is seen.

Please do not get this backwards. See Section [4.12.4 In-line assembly code](#).

**once**: The option **+pragma(*identifier*,once)** allows the programmer to establish an arbitrary *identifier* (usually the identifier **once**) as an indicator that the header is to be included just once. To mimic the Microsoft C++ compiler use the option: **+pragma(once,once)**. Then, a subsequent appearance of the pragma:

```
#pragma once
```

within a header will cause that header to be included just once. Subsequent attempts to include the header within the same module will be ignored.

**message**: The option **+pragma(*identifier*, message)** allows the programmer to establish an arbitrary *identifier* (usually the identifier **message**) as a pragma that will produce a message to standard out. For example:

```
+pragma(message,message)
```

will cause the pragma:

```
#pragma message "hello from file" __FILE__
```

to write to standard out a greeting identifying the file within which the pragma is contained. As this example shows, macros are expanded as encountered in the message line. Also, messages fall under control of the conditional compilation statements, `#if`, etc. Following the Microsoft compiler, if the first token is a left parenthesis then only the parenthetical expression will be output. Other information on the line is not output.

**ppw:** The option `+pragma( identifier, ppw )` will endow identifier with the `ppw` pragma action, which means reprocess the line as a preprocessor statement but with the word "pragma" ignored. Thus:

```
//lint +pragma( include, ppw )
#pragma include "abc.h"
```

will give the pragma keyword "include" the pragma action `ppw`, which will cause the next line to operate just like a standard `#include` preprocessor line.

**macro, fmacro and options:** Pragmas provide an avenue for the programmer to communicate to a compiler that is not governed by the syntax of the language. In most cases PC-lint Plus can ignore this information. Occasionally, however, the programmer will want us to act on this information so that he, the programmer, is not inserting the information twice, once for his compiler and once for PC-lint Plus.

Obviously it is impossible to provide compatible pragma recognition for all compilers now and into the future. The best general method of handling this seems to be to convert the pragma into a macro. Through the macro definition process, the macro can then become whatever the programmer wants. In particular it can become a `/*lint options...*/` comment and thereby be converted into a PC-lint Plus option. Alternatively it can be converted into code.

There are three basic ways of converting a pragma into a macro; these are identified as pragma types `macro`, `fmacro` and `options`.

**macro:** If a pragma is identified as `macro` as in the option

```
+pragma( identifier, macro )
```

then when a pragma by that name is encountered, the name is prefixed with the string 'pragma\_' and all the information to the right of the *identifier* is enclosed in parentheses. For example, one compiler supports statements such as:

```
#pragma port x @ 0x100
```

This would identify `x` as a particular I/O port. Later in the program there may be assignments to or from `x`. If PC-lint Plus were to ignore the pragma, it would have to emit syntax errors on every use of `x`.

If the option `+pragma( port,macro )` is given, the above pragma will be converted into:

```
pragma_port( x @ 0x100 )
```

Presumably there is a macro definition that resembles:

```
#define pragma_port( s ) volatile unsigned s;
```

Such a definition can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

**fmacro:** The second form of macroizing a pragma is identified as `fmacro` (meaning function macro). This would be used in a `+pragma` option having the form:

```
+pragma( identifier, fmacro )
```

With the `fmacro` type, instances of the pragma are assumed to already be in functional notation. Like the macro type, the name of the pragma is prefixed with `pragma_` to avoid conflicts with other uses of the pragma name. Aside from this prefixing the pragma is taken as found in the pragma statement and employed as a macro. For example, consider a pragma called `warnings`, which appears as:

```
#pragma warnings(no)
```

or

```
#pragma warnings(yes)
```

Presumably the `no` form turns off warnings and the `yes` form turns them back on.

If the option `+pragma(warnings,fmacro)` is given, the first pragma will be converted into:

```
pragma_warnings(no)
```

and the second will be converted into:

```
pragma_warnings(yes)
```

The programmer will want to provide a set of macros that can convert these pragmas into the equivalent form for PC-lint Plus. This could be done as follows:

```
#define pragma_warnings(x) pragma_warnings_##x
#define pragma_warnings_no /*lint -save -wl */
#define pragma_warnings_yes /*lint -restore */
```

These macro definitions can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

**options:** The third form of pragma that can be macroized is identified as '`options`' using an option of the form:

```
+pragma( identifier, options )
```

When a pragma having the name *identifier* is used, it is presumably followed by a blank-separated sequence of options having the form *name=value*. An example is:

```
#pragma OPTIONS tab=4 length=80
```

In this form, each individual option becomes a potential macro invocation. The name of this macro is formed by concatenating `pragma_`, the *identifier*, which in this case is `OPTIONS`, followed by underscore and the name of the suboption. Thus for the suboption `tab` the name of the macro is `pragma_OPTIONS_tab`. As an example, suppose the option `+pragma( OPTIONS, options )` is given and the following macro defined.

```
#define pragma_OPTIONS_tab(x) /*lint -t##x */
```

Then the above pragma will result in just the single macro invocation:

```
pragma_OPTIONS_tab(x)
```

This will invoke the `tab` option of PC-lint Plus. The '`length=80`' option is ignored. To attach meaning to the `length` option it would only be necessary to define a macro whose name would be `pragma_OPTIONS_length`.

**include\_alias:** The option `+pragma(identifier, include_alias)` allows the programmer to indicate an arbitrary identifier as mimicking the Microsoft C/C++ compiler pragma `include_alias`. For example:

```
+pragma(alias_include, include_alias)
```

will cause the pragmas:

```
#pragma alias_include( "a.h", "123.h" )
```

to inform Lint, when a `#include` directive appears for `"a.h"`, to search for `"123.h"` instead.

**identifier:** This option `-pragma( identifier )` will remove the pragma whose name is *identifier*. Thus:

```
-pragma(message)
```

will remove the `message` pragma.

#### 4.12.8 Arbitrary Width Integer Types

Integer types with an arbitrary number of bits (between 2 and 65,536 bits for signed types and 1 and 65,536 bits for unsigned types) are supported as exact-width types using the type specifier `_BitInt(n)` where *n* is the number of bits used to represent the type. `_BitInt` types are signed by default and may be combined with the `signed` or `unsigned` keywords. For example, the following will define `uint40_t` as a typedef for an unsigned 40-bit type:

```
typedef unsigned _BitInt(40) uint40_t;
```

The `-d` option can be used to define compiler-specific keywords as macros that expand to the equivalent `_BitInt` type, e.g. `-d_bit="unsigned _BitInt(1)"`.

Although this type specifier is part of the C23 language standard, its use will be allowed in analysis of any C or C++ language mode.

## 5 Libraries

Please note: This chapter is not about how to include header files that may be in some directory other than the current directory. For that information see the `-i` option (Section 4.4.2 [Preprocessor](#)) or Section 18.2.1 [INCLUDE Environment Variable](#). This chapter explains how information in header files (and possibly modules) is interpreted.

Examples of libraries are compiler libraries such as the standard I/O library, and third-party libraries such as windowing libraries, and database libraries. Also, an individual programmer may choose to organize a part of his own code into one or more libraries if it is to be used in more than one application. The important features of libraries, in so far as linting is concerned, are:

- (a) The source code is usually not available for linting.
- (b) The library is used by programs other than the one you are linting.

Therefore, to produce a full and complete analysis it is essential to know which headers represent libraries. It is also possible for modules to be available for linting but, because they are created beyond the control of the immediate programmer, they too can benefit from the designation 'library'.

### 5.1 Library Header Files

A library header file is a header file that describes (in whole or in part) the interface to a library.

The most familiar example of a library header file is `stdio.h`. Consider the file `hello.c`:

```
#include <stdio.h>

int main(void) {
    printf( "hello world\n" );
}
```

Without the header file, PC-lint Plus would complain that `printf` was neither declared (Informational 718) nor defined (Warning 526). (The distinction between a declaration and a definition is extremely important in C/C++. A definition for a function, for example, uses curly braces and there can be only one of them for any given function. Conversely, a declaration for a function ends with a semi-colon, is simply descriptive, and there can be more than one).

If `hello.c` were a C++ program an even stronger message would be issued, but we will assume a straight C program.

With the inclusion of `stdio.h` (assuming `stdio.h` contains a declaration for `printf`), PC-lint Plus will not issue message 718. Moreover, if `stdio.h` is recognized as a library header file, (it is by default because it was specified with angle brackets), PC-lint Plus will understand that source code for `printf` is not necessarily available and will not issue warning 526 either. Note: Other messages associated with library headers are not suppressed automatically. But you may use `-wlib` or any of the `-elib...` options for this purpose. See Section 4.3.1 [Error Inhibition](#).

A header file can become a library header file if:

- (a) It falls within one of the four broad categories of the option `+libclass`, viz. `all`, `ansi`, `angle` and `foreign` (described below), and is not excluded by either the `-libdir` or the `-libh` option.
- (b) OR, for finer control, it comes from a directory specified with `+libdir` and is not specifically excluded with `-libh`.
- (c) OR, for the finest control, it is specifically included by name via `+libh`.

(d) OR, is included within a library header file.

You may determine whether header files are library header files by using some variation of the `-vf` verbosity option. For each included library header you will receive a message similar to:

```
Including file c:\compiler\stdio.h (library)
```

The tag: `'(library)'` indicates a library header file. Other header files will not have that tag.

What follows is a more complete description of the three options used to specify if or when a header file is a library header file.

`+libclass( identifier [, identifier] ...)` specifies the set or sets of header files that are assumed to be library header files.

Each identifier can be one of:

**angle** All headers specified with angle brackets.

**foreign** All header files found in directories that are on the search list (`-i` or `INCLUDE` as appropriate).

Thus, if the `#include` contains a complete path name then the header file is not considered 'foreign'. To endow such a file with the library header property use either the `+libh` option or angle brackets. For example, if you have

```
#include "\include\graph.h"
```

and you want this header to be regarded as a library header use angle brackets as in:

```
#include <\include\graph.h>
```

or use the option:

```
+libh(\include\graph.h)
```

Similar remarks can be made about

```
#include "include\graph.h"
```

If a search list (specified with `-i` option or `INCLUDE`) is used to locate this file it is considered **foreign**; otherwise it is not.

**ansi** The 'standard' ANSI/ISO C header files, viz.

```
assert.h    limits.h    stddef.h
ctype.h     locale.h    stdio.h
errno.h     math.h      stdlib.h
float.h     setjmp.h    string.h
fstream.h   signal.h    strstream.h
iostream.h  stdarg.h    time.h
```

**all** All header files are regarded as being library headers.

By default, `+libclass(angle,foreign)` is in effect. This option is not cumulative. Any `+libclass` option completely erases the effect of previous `+libclass` options. To specify no class use the option `+libclass()`.

`+libdir(directory [, directory] ... )` activates

`-libdir(directory [, directory]...)` deactivates

the directory (or directories) specified. The notion of *directory* here is identical to that in the `-i` option. If a *directory* is activated then all header files found within the directory will be regarded as library header files (unless specifically inhibited by the `-libh` option). It overrides the `+libclass` option for that particular directory. For example:

```
+libclass()
+libdir( c:\compiler )
+libh( os.h )
```

requests that no header files be regarded as library files except those coming from directory `c:\compiler` and the header `os.h` (see below). Also,

```
+libclass( foreign )
-libdir( headers )
```

requests that all headers coming from any foreign directory except the directory specified by `headers` should be regarded as library headers.

Wild card characters '\*' and '?' are supported. Note: A file specified as

```
#include "c:\compiler\i.h"
```

is not regarded as being a library header even though `+libdir(c:\compiler)` was specified. Only files found in `c:\compiler` via a search list (`-i` or `INCLUDE`) are so regarded and only when the `-i` option matches the `libdir` parameter. For example,

```
#include "compiler\i.h"
```

will also not be considered as library even though the `-ic:` option is given, and the file is found by searching. The `-i` search directory (`c:`) is not matching the `libdir` directory (`c:\compiler`).

`+libh( file[, file] ... )` adds  
`-libh( file[, file]...)` removes

files from the set that would otherwise be determined from the `+libclass` and `-/+libdir` options. For example:

```
+libclass( ansi, angle )
+libh( windows.h, graphics.h )
+libh( os.h ) -libh( float.h )
```

requests that the header files described as `ansi` or `angle` (except for `float.h`) and the individual header files: `windows.h`, `graphics.h` and `os.h` (even if not specified with angle brackets) will be taken to be library header files.

Wild card characters '\*' and '?' are supported.

For `libh` to have an effect, its argument must match the string between quotes or angle brackets in the `#include` line. Thus in the case of:

```
#include <../lib/graphics.h>
```

you must have `+libh(../lib/graphics.h)`.

Note that the `libh` option is accumulative whereas the `+libclass` option overrides any previous `+libclass` option including the default.

When a `#include` statement is encountered, the name that follows the `#include` is defined to be the *header-name* (even if the name is a compound name containing directories). When an attempt is made to open the file, a list of directories is consulted, which are all those specified by `-i` options and the `INCLUDE` environment variable. The directory that is used to successfully open the file is defined to be

the *header-directory*.

The options `+libdir(...)` and `-libdir(...)` are applied to the *header-directory* and the options `+libh(...)` and `-libh(...)` are applied to the *header-name* (as defined in the previous paragraph). For example, given the following:

```
#include "graphics\shapes.h"
```

Suppose that the following option had been given:

```
-iC:\
```

and suppose further that a file `"C:\graphics\shapes.h"` exists. Then the *header-name* would be `"graphics\shapes.h"` and the *header-directory* would be `"C:\"`. Any one of the following options could be used to designate the file as a library file.

```
+libh( graphics\* )
+libh( *shapes.h )
+libdir( C:* )
+libdir( C:\ )
```

## 5.2 Library Modules

You may designate that a module is a library module using the option:

```
+libm( module-name )
```

You would normally use just the '+' form of the option. But you may use `-libm` to undo the effects of a `+libm` option with some arguments.

This option has the effect of designating the entire module and all of the header files that it includes, as "library". That is, messages will be inhibited via `-wlib` or `-elib...` options. Unused globals defined within such a module will draw no complaints, etc.

As an example, suppose you have an application `alpha.c`, and that this code requires the services of a module `beta.c`, which is generated by a separate program. Typically the interface to `beta.c` will be described by a header file `beta.h` and a typical linting can be specified by:

```
lint +libh( beta.h ) alpha.c
```

But another possibility is to include `beta.c` in the lint. This would have the advantage of facilitating inter module value tracking. The typical command to do this would be:

```
lint +libh( beta.h ) +libm( beta.c ) alpha.c beta.c
```

Note that the option `libm` takes a pattern that may include wild-cards. Let us suppose that our generator will generate not just `beta.c` but a sequence of three modules

```
beta1.c  beta2.c  beta3.c
```

Then they can all be designated as library with the single option

```
+libm( beta*.c )
```

## 5.3 Assembly Language Modules

In this section we deal with the case of assembly-language modules. For in-line assembly code see [Section 4.12.4 In-line assembly code](#).

If one or more modules of your application are written in assembly language or, equivalently, in some language other than C or C++ (a common phrase is "mixed language"), you must arrange so that the missing code



does not cause PC-lint Plus to give spurious messages. The most common way of proceeding is to create a header file describing the assembly language portion of your application. This header file, say `asm.h`, will have property (a) of library header files in that the objects declared therein will not be defined in files seen by PC-lint Plus. Hence we make it a library header file with the option:

```
+libh(asm.h)
```

Finally, the assembly language portion of your application may be the only portion of your application that is referencing, initializing or accessing some variable or function. A spurious "not referenced" or "not accessed" message would be given. The easiest thing to do is to explicitly suppress the message(s). For example, if the assembly language portion is the only portion accessing variable `alpha` and you are getting message 552, then place option `-esym(552,alpha)` among your lint options. If you are using our suggested setup, as described in Section 19.2 Recommended Setup, then `std.lnt` will now have the contents:

```
c.lnt
options.lnt
+libh(asm.h)
-esym(552,alpha) //accessed in assembly language
```

You might be tempted to place these options in lint comments within `asm.h`. Unfortunately, the `libh` option will be set too late to establish `asm.h` as a library header.

You might yet say that "My assembly language routines are sometimes opted out and sometimes opted in, and this is under control of a global preprocessor variable `USEASM`. When opted out, C/C++ equivalent routines are activated. How can I cope with this varying situation?"

This actually makes the situation easier. Just make sure that when you are linting, `USEASM` is opted out. You might use:

```
#ifdef _lint
#undef USEASM
#endif
```

or some equivalent sequence. In this way, lint will know the intent of the assembly code from the equivalent C/C++ code. The previously suggested options of `+libh` and `-esym` are then not necessary.

## 6 Precompiled Headers

### 6.1 Introduction to precompiled headers

Most readers of this information will already be familiar with the notion of a precompiled header. With traditional precompiled headers, a single header is designated as one to be precompiled. In PC-lint, such a file was one that was expected to be `#include`'d in the source module. In PC-lint Plus, an existing precompiled header is processed as a prefix header and is loaded before each module that follows the `pch` option designating the PCH header, regardless of whether that module explicitly `#includes` the header or not. PC-lint Plus's precompiled headers act as a sort of "on disk caching" of a previously compiled header file. Information is loaded into memory as needed, reducing processing time.

Two types of precompiled headers exist; those precompiled for C modules have the extension "lcph" while those precompiled for C++ modules have the extension "lpph". For example, if a header's name is "a.h" and it is being precompiled for a C module, the resultant file will be "a.lcph". If a specific header is marked for precompilation and a precompiled version does not already exist, PC-lint Plus will precompile that header at the start of examining a module, saving it to a file with the relevant extension. PC-lint Plus will create a C or C++ version of the precompiled header only if necessary to process the next module. Otherwise, the file will be opened and read in as needed.

Note: if you `#include` the file from which a precompiled header is created, you are advised to follow what is typically considered good programming practice and make sure that header contains a standard include guard.

To designate that a header is to be precompiled use the option:

```
-pch( header-name )
```

The *header-name* should be the name of a file that can be found via the traditional include process, such as by examining `-i` options.

If you want to use a precompiled header for some modules and not for others, you can disable the use of a precompiled header with the option:

```
-pch()
```

For example, to use header "a.h" as a precompiled header for the first three modules of your project and not the fourth, the arguments passed to PC-lint Plus should look something like this:

```
-pch(a.h)
module1.c      // creates `a.lcph` if needed
module2.cpp    // creates `a.lpph` if needed
module3.c      // `a.lcph` already exists; load as needed
-pch()
module4.c      // neither create nor use any precompiled header
```

You can also specify multiple precompiled headers per project, though only one per module. You do so by passing another `-pch(header-name)` styled option to PC-lint Plus after the previous module name and before the next. If we alter our example above to use a precompiled header for "b.h" in the fourth module of the project, the argument list would look something like this:

```
-pch(a.h)
module1.c      // creates `a.lcph` if needed
module2.cpp    // creates `a.lpph` if needed
module3.c      // `a.lcph` already exists; load as needed
-pch(b.h)
module4.c      // creates `b.lcph` if needed
```

## 6.2 Designating the precompiled header

To designate that a header is to be precompiled use the option:

```
-pch( header-name )
```

The *header-name* should be that name used between angle brackets or between quotes on the `#include` line. In particular, if the name on the `#include` line is not a full path name do not use a full path name in the option.

Normally a precompiled header is the first header encountered in each of the modules that include it. Occasionally it is not, because the `-header()` option forcefully (if silently) includes a header just prior to the start of each module. Also, it just might be desirable to include a header prior to the one declared to be the precompiled header. So earlier headers are permitted. But if a precompiled header does follow an include sequence, it must follow that same include sequence in every module in which it is included. Otherwise a diagnostic will be issued.

## 6.3 Monitoring precompiled headers

The sequence of events that takes place when a precompiled header is included can be monitored by using a variant of the verbosity option that contains or implies the letter 'f'. Given the option sequence:

```
-pch(x.h) -vf
```

we would expect to see, at the first time `x.h` is included, the verbosity line:

```
Including file x.h (bypass)
```

As indicated above, `x.h` becomes, of necessity, a file to be bypassed in subsequent modules. After fully processing `x.h` and all of its includes we will see the line:

```
Outputting to file x.lph
```

The extension `"lph"` stands for "lint precompiled header". The name of the file containing the precompiled output is formed by appending this extension onto the root of the file named in the `pch` option.

In subsequent modules you will see the verbosity line:

```
Bypassing x.h
```

in place of a line that would normally show an include of this header.

If the program were to be linted subsequently with the same options, then instead of seeing a verbosity line indicating that `x.h` were included and `x.lph` were written we would see:

```
Absorbing file x.lph
```

reflective of the fact that `x.lph` contains binary information representative of the information in `x.h`.

## 6.4 The use of make files

The `.lph` file is not automatically regenerated when the original header (or any of its sub headers) is modified. If it is important that it must be done automatically then you will need a `make` facility or its equivalent. An entry in the `make` file could be as simple as:

```
x.lph: x.h ...
      del x.lph
```

In words, an `x.lph` is composed of `x.h` plus any of its included header files and is 'manufactured' by a deletion of `x.lph`. If this confuses the `make` facility then you might try something like:

```
request.lph: x.h ...
            del x.lph
            touch request.lph
```

Here you need to create a file called "**request.lph**" whose content is the minimal necessary for **make** to consider it a file. Whenever any of a collection of headers is modified, **x.lph** is deleted and the date of the **request.lph** is updated.

## **6.5 Lint comment options and precompiled headers**

Lint comment options residing in headers are not currently preserved in the generated precompiled headers. Options, such as suppressions, appearing within headers that are used as precompiled headers (or included by such headers) should therefore be moved out of the header to achieve the desired effect (parameterized suppression options are particularly useful here).

## 7 Strong Types

*Strong type checking is gold  
Normal type checking is silver  
But casting is brass*

### 7.1 Rationale

The strong type system allows you to imbue typedefs with flexible type-checking properties and can perform dimensional analysis. For example, consider the law of universal gravitation:

$$F = G \frac{m_1 m_2}{r^2}$$

The following code attempts to implement this, but contains a mistake:

```

1  typedef double Meter, Second, Velocity, Acceleration;
2  typedef double Kilogram, Newton;
3  typedef double Area, Volume;
4  typedef double GravitationalConstant;
5
6  const GravitationalConstant G = 6.67e-11;
7
8  Newton attraction(Kilogram mass1, Kilogram mass2, Meter distance) {
9      return (mass1 * mass2) / (distance * distance);
10 }
```

A compiler is not interested in (and has no obligation to warn you about) the dimensional mismatch here caused by forgetting to multiply by G. Running this example through lint with the appropriate strong type options produces the following messages:

```

strong type mismatch: assigning '(Kilogram*Kilogram)/(Meter*Meter)' to 'Newton'
did you mean to multiply by a factor of type 'GravitationalConstant'?
```

If you are curious what options were used to get these units into the strong type system, see [Full Source for the Gravitation Example](#).

### 7.2 Creating Strong Types with `-strong`

The primary option used to interact with the strong type system is

```
-strong(flags[,name ...])
```

This option identifies each *name* as a strong type with properties specified by *flags*. Presumably there is a later `typedef` defining any such *name* to be a type. If no *name* is provided, the specified *flags* will be taken as the default for types without explicit `-strong` options. Flags are uppercase letters that indicate some aspect of a type's behavior, and they can be modified by following them with softeners. Softeners are represented using lowercase letters and must immediately follow the flag they are modifying.

- A Check strong types on Assignment. Issue a warning upon assignment (where assignment refers to using the assignment operator, returning a value, passing an argument or initializing a variable). A may be followed by one or more softening modifiers:
  - i ignore Initialization
  - r ignore Return statements
  - p ignore Passing arguments
  - a ignore the Assignment operator
  - c ignore assignment of Constants (literals)
  - z ignore assignment of integer constant expressions equal to Zero (non-strong casts are ignored)
- X Check strong types on eXtraction. This flag issues warnings in the same contexts as the A flag, but checks on behalf of the value being assigned. The softeners for A cannot be used with X.
- J Check strong types when Joining two operands of a binary operator. J may be followed by one or more of the following modifiers:
  - e ignore Equality operators, (== and !=), and the conditional operator, (?:)
  - r ignore Relational operators, (>, >=, <, and <=)
  - m ignore Multiplicative operators, (\*, /, and %)
  - d indicates that this strong type is a Dimension (see [7.4.1 Dimensional Types](#))
  - n indicates that this strong type is dimensionally Neutral (see [7.4.2 Dimensionally Neutral Types](#))
  - a indicates that this strong type is Antidimensional (see [7.4.3 Antidimensional Types](#))
  - o ignore Other (non-multiplicative) binary operators, (+, -, |, &, and ^)
  - c ignore combining with constants
  - z ignore combining with Zero, as in Az above
- B Designate a major boolean type. Only one boolean type may exist whether it comes from the B or b flag. The result of all boolean operators will be a value compatible with this type. Contexts that expect a boolean value will require their operands to be of the major boolean type.
- b Designate a minor boolean type. Only one boolean type may exist whether it comes from the B or b flag. The result of all boolean operators will be a value compatible with this type. This flag places no requirement on the values used in contexts that expect a boolean, in contrast to B.
- l Designate a type as inherently compatible with library functions. This includes assignment from library function return values and as library function arguments.
- f Indicates bit-fields of length one are not automatically boolean (by default they are). This is a modifier that can only accompany one of the boolean flags (either B or b above). .

## 7.3 Strong Types for Array Indices

### Description

```
-index( flags, ixtype, sitype [, sitype ...] )
```

This option is supplementary to and can be used in conjunction with the `-strong` option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype*

(or *sitype*'s if more than one is provided). Please note: both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a **typedef** declaration. *flags* can be

- c allow Constants as well as *ixtype*, to be used as indices.
- d allow array Dimensions to be specified without using an *ixtype*.

### Examples of -index

For example:

```
//lint -strong( AzJcX, Count, Temperature )
//lint -index( d, Count, Temperature )
//      Only Count can index a Temperature

typedef float Temperature;
typedef int Count;
Temperature t[100];      // OK because of d flag
Temperature *pt = t;     // pointers are also checked

                        // ... within a function
Count i;

t[0] = t[1];             // Warnings, no c flag
for( i = 0; i < 100; i++ )
t[i] = 0.0;              // OK, i is a Count
pt[1] = 2.0;             // Warning
i = pt - t;              // OK, pt-t is a Count
```

In the above, **Temperature** is said to be *strongly indexed* and **Count** is said to be a *strong index*.

If the **d** flag were not provided, then the array dimension should be cast to the proper type as for example:

```
Temperature t[ (Count) 100 ];
```

However, this is a little cumbersome. It is better to define the array dimension in terms of a manifest constant, as in:

```
#define MAX_T (Count) 100
Temperature t[MAX_T];
```

This has the advantage that the same **MAX\_T** can be used in the **for** statement to govern the range of the **for**.

Note that pointers to the Strongly Indexed type (such as **pt** above) are also checked when used in array notation. Indeed, whenever a value is added to a pointer that is pointing to a strongly indexed type, the value added is checked to make sure that it has the proper strong index.

Moreover, when strongly indexed pointers are subtracted, the resulting type is considered to be the common Strong Index. Thus, in the example,

```
i = pt - t;
```

no warning resulted.

It is common to have parallel arrays (arrays with identical dimensions but different types) processed with similar indices. The **-index** option is set up to conveniently support this. For example, if **Pressure** and **Voltage** were types of arrays similar to the array **t** of **Temperature** one might write:

```
//lint -index( , Count, Temperature, Pressure, Voltage )
...
Temperature t[MAX_T];
Pressure p[MAX_T];
Voltage v[MAX_T];
...
```

### Multidimensional Arrays

The indices into multidimensional arrays can also be checked. Just make sure the intermediate type is an explicit `typedef` type. An example is `Row` in the code below:

```
/* Types to define and access a 25x80 Screen.
a Screen is 25 Row's
a Row is 80 Att_Char's */

/*lint -index( d, Row_Ix, Row )
-index( d, Col_Ix, Att_Char ) */

typedef unsigned short Att_Char;
typedef Att_Char Row[80];
typedef Row Screen[25];

typedef int Row_Ix; /* Row Index */
typedef int Col_Ix; /* Column Index */

#define BLANK (Att_Char) (0x700 + ' ')

Screen scr;
Row_Ix row;
Col_Ix col;

void main()
{
    int i = 0;

    scr[ row ][col ] = BLANK;      /* OK */
    scr[ i ][ col ] = BLANK;      /* Warning */
    scr[col][row] = BLANK;      /* Two Warnings */
}
```

In the above, we have defined a `Screen` to be an array of `Row`'s. Using an intermediate type does not change the configuration of the array in memory. Other than for type-checking, it is the same as if we had written:

```
typedef Att_Char Screen[25][80];
```

## 7.4 Dimensional Analysis

Unlike other binary operators that expect their operands to agree in strong type, multiplication and division often can and should handle different types in what is commonly referred to as dimensional analysis. But not all strong types are the same in this regard. The strong type system recognizes three different kinds of treatment with regard to multiplication and division.



### 7.4.1 Dimensional Types

A *dimension* is a strong type such that when two expressions are multiplied or divided and each type is a dimension, then the resulting type will also be a dimension whose name will be a compound string representing the product or quotient of the operands (reduced to lowest terms). The modulus operator % will have a resultant type equal to the type of the numerator.

For example:

```
//lint -strong( AJdX, Sec )
typedef double Sec;
Sec x, y;
...
x = x * y;      // warning: '(Sec*Sec)' is assigned to 'Sec'
y = 3.6 / x;    // warning: '1/Sec' is assigned to 'Sec'
```

Flags 'AJdX' contain the Join phrase 'Jd' designating that Sec is a dimension. Strictly speaking the 'd' is not necessary because the normal default is to make any strong type dimensional. However, there is a flag option **-fdd** (turn off the Dimension by Default flag), which will reverse this default behavior, so it is probably wise to place the 'd' in explicitly.

Dimensional types are treated in greater detail later.

### 7.4.2 Dimensionally Neutral Types

A *dimensionally neutral* type is a strong type such that when multiplied or divided by a dimension will act as a non-strong type.

For example:

```
//lint -strong( AJdX, Sec )
typedef double Sec;

//lint -strong( AJnX, Cycles )
typedef double Cycles;
Cycles n;
Sec t;
...
t = n * t;      // OK, Cycles are neutral
t = t / n;      // still OK.
n = n / t;      // warning: '1/Sec' assigned to 'Cycles'
```

The **n** softener of the J flag as in the AJnX sequence above designates that type **Cycles** is dimensionally neutral and will drop away when combined multiplicably with the dimension **Cycles** as shown in the first two assignments. However, **Cycles** acts as a strong type in every other regard. An illustration of this is the last line in this example, which produces a warning that the type '1/Sec' is being assigned to **Cycles**.

Thus, **Cycles** is playing the role that it traditionally plays in Physics and Engineering. It contains no physical units and when multiplied or divided by a dimension does not change the dimensionality of the result.

### 7.4.3 Antidimensional Types

An *antidimensional* type is a strong type that when multiplied or divided is expected to be combined with the same type, or one that is compatible through the usual strong type hierarchies. It functions in this regard much like addition and subtraction.

For example:

```
//lint -strong( AJaX, Integer )
typedef int Integer;
Integer k;
int n;
...
k = k * k; // OK
k = n * k; // warning: Integer joined with non-Integer
```

The sequence `Ja` in the above indicates that `Integer` is antidimensional.

#### 7.4.4 Multiplication and Division of Dimensional Types

The strong type mechanism can support the traditional dimensional analysis exploited by physicists, chemists and engineers. When strong types are added, subtracted, compared or assigned, the strong types need merely match up with each other. However, multiplication and division can join arbitrary dimensional types and the result is often a new type. Consider forming the velocity from a distance and a time:

```
//lint -strong( AcJcX, Met, Sec, Velocity = Met/Sec )
typedef double Met, Sec, Velocity;
Velocity speed( Met d, Sec t )
{
    Velocity v;
    v = d / t;           // ok
    v = 1 / t;           // warning
    v = (3.5/t) * d;     // ok
    v=(1/(t*t))*d*t;     // ok
    return v; // ok
}
```

In this example, the 4th argument to the `-strong` option:

```
Velocity = Met/Sec
```

relates strong type `Velocity` to strong types `Met` and `Sec`. This particular suboption actually creates two strong types: `Velocity` and `Met/Sec` and relates the two types by making `Met/Sec` the parent type of `Velocity`. This relationship can be seen in the output obtained from the option `-vh` (or the compact form `-vh-`). As an example the results of the `-vh` option for the above example are:

```
- Met
- Sec
- Met/Sec
|
+ - Velocity
- 1/Sec
- (Sec*Sec)
- 1/(Sec*Sec)
- Met/(Sec*Sec)
```

The division of `Met` by `Sec` (within the option) can be produced in many equivalent ways. E.g.

```
Velocity = (1/Sec) * Met
Velocity = ((1/Sec) * (Met))
Velocity = (Met/(Sec*Sec)) * Sec
```

are all equivalent. All of these dimensional expressions are reduced to the canonical form `Met/Sec`, which was the form given in the original option. Note that parentheses can be used freely and in some cases must be used to obtain the correct results. E.g.

```
Acceleration = Met/Sec*Sec      // wrong
Acceleration = Met/(Sec*Sec)    // correct
```

We follow C syntactic rules where the operators bind left to right and the example labeled 'wrong' results, after cancellation, in just **Met**.

Briefly and for the record the canonical form produced is:

$$(F1 * F2 * \dots * F_n) / (G1 * G2 * \dots * G_m)$$

where each  $F_i$  and each  $G_i$  are simple single-identifier sorted strong types and where  $n \geq 0$  and  $m \geq 0$  but if  $n$  is less than 2 the upper parentheses are dropped out and if  $m$  is less than 2 the lower parentheses are dropped and if  $n$  is 0 the numerator is reduced to 1 and if  $m$  is 0 the entire denominator including the  $/$  is dropped.

Returning to our original example (the function **speed**), when the statement:

```
v = d/t;
```

is encountered and an attempt is made to evaluate  $d/t$  the dimensional nature of the types of the two arguments is noted and the names of these types is combined by the division operator to produce "**Met/Sec**". This uses essentially the same algorithms and canonicalization as the compound type analysis with a **-strong** option. The resulting type is assigned to **Velocity** without complaint because of the previously described parental relationship that exists between these two strong types.

In the next statement

```
v = (3.5/t) * d;
```

the division results in the creation of a new strong type (**1/Sec**), which when multiplied by **Met** will become **Met/Sec**. The created type will have properties **AJcdX** and the underlying type will be the type that a compiler would compute.

#### 7.4.5 Dimensional Types and the % operator

Let's say you have a paper 400 lines long and the printing requires 60 lines/page. How many full pages will we require? The answer is

```
400 lines / (60 lines/page)
= 6 pages
```

How many lines are left over? The answer is

```
400 lines % (60 lines/page)
= 40 lines
```

Thus, unlike division, the **%** operator yields a dimension that equates to the dimension of the numerator (in this case, **lines**) while ignoring the dimension of the 2nd operand.

#### 7.4.6 Conversions

A simple example in the use of Dimensional strong types is that of providing a fail-safe method of converting from one system of units to another. Such conversions can quite often be accomplished by a single numeric factor. Such conversion factors should have dimensions attached to prevent mistakes. E.g.

```
// Centimeters to/from Inches
//lint -strong( AJdX, In, Cm, CmPerIn = Cm/In )
typedef double In, Cm, CmPerIn;
CmPerIn cpi = (CmPerIn) 2.54;    // conversion factor
void demo( In in, Cm cm )
{
    ...
}
```

```

    in = cm / cpi;                // convert cm to in
    ...
    cm = in * cpi;                // convert in to cm
    ...
}

```

In this example we are defining a conversion factor, `cpi`, that will allow us to convert inches to centimeters (by multiplication) and convert centimeters to inches (via division). Without strong types, conversion factors can be misused. Do I multiply or divide? Using strong types you can be assured of getting it right.

Obviously not all conversions fall into the category of being described by a conversion factor. Conversions between Celsius and Fahrenheit, for example, require an expression and this typically means defining a pair of functions as in the following:

```

//lint -strong( AJdX, Fahr, Celsius )
typedef double Fahr, Celsius;
Celsius toCelsius( Fahr t )
    { return (t-(Fahr)32.) * (Celsius)5. / (Fahr)9.; }
Fahr toFahr( Celsius t )
    { return (Fahr)32. + t * (Fahr)9. / (Celsius)5.; }

```

The function call overhead is probably not significant, but if it is, you may declare the functions to be inline in C++. Some C systems support inline functions, but in any case, you can use macros.

Now let us suppose a confused programmer had written:

```

Fahrenheit f;
Celsius c;
...
f = toCelsius (c);    // Type Violations

```

Then there would be two strong type violations since passing `c` to a `Fahrenheit` variable is bad as is assigning a `Celsius` value to `f`.

#### 7.4.7 Integers

Although the examples of dimensional analysis offered above refer to floating point quantities, the same principles apply to integer arithmetic. E.g.

```

#include <stdio.h>
#include <limits.h>
//lint -strong( AcJdX, Bytes, Bits )
//lint -strong( AcJdX, BitsPerByte = Bits / Bytes )
typedef size_t Bytes, Bits, BitsPerByte;
BitsPerByte bits_per_byte = CHAR_BIT;
Bytes size_int = sizeof(int);
Bits length_int = size_int * bits_per_byte;

```

In this example `Bits` is the length of an object in bits and `Bytes` is the length of an object in bytes. `bits_per_byte` becomes a conversion factor to translate from one unit to the other. The example shows the use of that conversion factor to compute the number of bits in an integer.

Let's say that you wanted to strengthen the integrity and robustness of a program by making sure that all shifts were by quantities that were typed `Bits`. For example you could define a function `shift_left` with the intention that this function have a monopoly on shifting `unsigned` types to the left. This could take the form:

```

inline unsigned shift_left( unsigned u, Bits b ) {
    return u << b;
}

```

```
}
```

A simple grep for "<<" can be used to ensure that no other shift lefts exist in your program. Note that the example deals only with `unsigned` but if there were other types that you wanted to shift left, such as `unsigned long`, you can use the C++ overload facility.

Using C you may also employ the `shift_left` function. However you may not have `inline` available and you may be concerned about speed. To obtain the required speed you can employ a macro as in:

```
#define Shift_Left(u,b) ((u) << (b))
```

But you will note that there is now no checking to ensure that the number of bits shifted are of the proper type. One approach is to use conditional compilation:

```
#ifdef _lint
#define Shift_Left(u,b) shift_left(u,b)
#else
#define Shift_Left(u,b) ((u) << (b))
#endif
```

This will work adequately in C. If the quantity being shifted is anything other than plain `unsigned`, you will need to duplicate this pattern for each type.

A probably better approach is to define a macro that can check the type, such as the macro `Compatible` defined below:

```
#ifdef _lint
#define Compatible(e,type) (*(type*)__Compatible = (e),(e))
static char __Compatible[100];
//lint -esym(528,__Compatible)
//lint -esym(551,__Compatible)
//lint -esym(843,__Compatible)
#else
#define Compatible(e,type) (e)
#endif
```

You could then define the original `Shift_Left` macro as:

```
#define Shift_Left(u,b) ((u) << compatible(b,Bits))
```

`Compatible(e,type)` works as follows. Under normal circumstances (i.e. when compiling) it is equivalent to the expression `e`. When linting it is also equivalent to `e` except that there is a side effect of assigning to some obscure array that has been artfully configured into resembling a data object of type `type`. A complaint will be issued if the expression `e` would draw a complaint when assigned to an object of type `type`.

In this way you can be assured that the shift amount is always assignment compatible with `Bits`. Note that there is no longer a need for the twin `Shift_Left` definitions. And `Compatible` can be used in many other places to assure that objects are typed according to program requirements.

For simplicity, we have focused on shifting left. Obviously, similar comments can be made for shifting right.

## 7.5 Strong Type Hierarchies

### 7.5.1 The Need for a Type Hierarchy

Consider a *Flags* type, which supports the setting and testing of individual bits within a word. An application might need several different such types. For example, one might write:

```
typedef unsigned Flags1;
typedef unsigned Flags2;
typedef unsigned Flags3;

#define A_FLAG (Flags1) 1
#define B_FLAG (Flags2) 1
#define C_FLAG (Flags3) 1
```

Then, with strong typing, an `A_FLAG` can be used with only a `Flags1` type, a `B_FLAG` can be used with only a `Flags2` type, and a `C_FLAG` can be used with only a `Flags3` type. This, of course, is just an example. Normally there would be many more constants of each *Flags* type.

What frequently happens, however, is that some generic routines exist to deal with *Flags* in general. For example, you may have a stack facility that will contain routines to push and pop *Flags*. You might have a routine to print *Flags* (given some table that is provided as an argument to give string descriptions of individual bits).

Although you could cast the *Flags* types to and from another more generic type, the practice is not to be recommended, except as a last resort. Not only is a cast unsightly, it is hazardous since it suspends type-checking completely.

### 7.5.2 The Natural Type Hierarchy

The solution is to use a type hierarchy. Define a generic type called `Flags` and define all the other *Flags* in terms of it:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef Flags Flags3;
```

In this case `Flags1` can be combined freely with `Flags`, but not with `Flags2` or with `Flags3`.

Hierarchy depends on the state of the `fhs` (Hierarchy of Strong types) flag, which is normally ON. If you turn it off with the

```
-fhs
```

option the natural hierarchy is not formed.

We say that `Flags` is a *parent* type to each of `Flags1`, `Flags2` and `Flags3`, which are its children. Being a parent to a child type is similar to being a base type to a derived type in an object-oriented system with one difference. A parent is normally interchangeable with each of its children; a parent can be assigned to a child and a child can be assigned to a parent. But a base type cannot normally be assigned to a derived type. But even this property can be obtained via the `-father` option (See Section 7.5.4 [Restricting Down Assignments \(-father\)](#)).

A generic *Flags* type can be useful for all sorts of things, such as a generic zero value, as the following example shows:

```
//lint -strong(AJX)
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
#define FZERO (Flags) 0
#define F_ONE (Flags) 1
```

```

void m()
{
  Flags1 f1 = FZERO;           // OK
  Flags2 f2;

  f2 = f1;                     // Warn
  if(f1 & f2)                   // Warn because of J flag
  f2 = f2 | F_ONE;             // OK
  f2 = F_ONE | f2;             // OK Flag2 = Flag2
  f2 = F_ONE | f1;             // Warn Flag2 = Flag1
}

```

Note that the type of a binary operator is the type of the most restrictive type of the type hierarchy (i.e., the child rather than the parent). Thus, in the last example above, when a `Flags` OR's with a `Flags1` the result is a `Flags1`, which clashes with the `Flags2`.

Type hierarchies can be an arbitrary number of levels deep.

There is evidence that type hierarchies are being built by programmers even in the absence of strong type-checking. For example, the header for Microsoft's Windows SDK, `windows.h`, contains:

```

...
typedef unsigned int    WORD;
typedef WORD            ATOM;
typedef WORD            HANDLE;
typedef HANDLE          HWND;
typedef HANDLE          GLOBALHANDLE;
typedef HANDLE          LOCALHANDLE;
typedef HANDLE          HSTR;
typedef HANDLE          HICON;
typedef HANDLE          HDC;
typedef HANDLE          HMENU;
typedef HANDLE          HPEN;
typedef HANDLE          HFONT;
typedef HANDLE          HBRUSH;
typedef HANDLE          HBITMAP;
typedef HANDLE          HCURSOR;
typedef HANDLE          HRGN;
typedef HANDLE          HPALETTE;
...

```

### 7.5.3 Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via `typedef` declaration has a limitation. All the types in a single tree must be the same underlying type. The `-parent` option can be used to supplement (or completely replace) the strong type hierarchy established via `typedef` declarations.

An option of the form:

```
-parent( Parent , Child [, Child] ... )
```

where *Parent* and *Child* are type names defined via `typedef` will create a link in the strong type hierarchy between the *Parent* and each of the *Child* types. The *Parent* is considered to be equivalent to each *Child* for the purpose of Strong type matching. The types need not be the same underlying type and normal checking between the types is unchanged.

A link that would form a loop in the tree is not permitted.

For example, given the options:

```
-parent(Flags1,Small)
-strong(AJX)
```

and the following code:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef unsigned char Small;
```

then the following type hierarchy is established:

```
      Flags
     /   \
Flags1   Flags2
  |
Small
```

If an object of type `Small` is assigned to a variable of type `Flags1` or `Flags`, no strong type violation will be reported. Conversely, if an object of type `Flags` or `Flags1` is assigned to type `Small`, no strong type violation will be reported but a loss of precision message will still be issued (unless otherwise inhibited) because normal type checking is not suspended.

If the `-fhs` option is set (turning off the hierarchy of strong types flag) a `typedef` will not add a hierarchical link. The only links that will be formed will be via the `-parent` option.

#### 7.5.4 Restricting Down Assignments (-father)

The option

```
-father( Parent , Child [, Child] ... )
```

is similar to the `-parent` option and has all the effects of the `-parent` option and has the additional property of making each of the links from *Child* to *Parent* one-way. That is, assignment from *Parent* to *Child* triggers a warning. You may think of `-father` as a strict version of `-parent`.

The rationale for this option is shown in the following example.

```
typedef int FIndex;
typedef FIndex Index;
```

Here `Index` is a special `Index` into an array. `FIndex` is a `Flag` or an `Index`. If negative, `FIndex` is taken to be a special flag and otherwise can take on any of the values of `Index`. By defining `Index` in terms of `FIndex` we are implying that `FIndex` is the parent of `Index`. The reader not accustomed to OOP may think that we have the derivation backwards, that the simpler `typedef`, `Index`, should be the parent. But `Index` is the more specific type; every `Index` is an `FIndex` but not conversely. Whereas it is expected that we can assign from `Index` to `FIndex` it could be dangerous to do the inverse.

Since we do not want down assignments we give the option

```
-father( FIndex, Index )
```

in addition to the strong options, say

```
-strong( AcJcX, FIndex, Index )
```

Then



```

FIndex n = -1;
Index i= 3;

i = n;          /* Warning */
n = i;          /* OK */

```

The safe way to convert a FIndex to Index is via a function call as in

```
Index F_to_I( FIndex fi )
```

## 7.6 Printing the Hierarchy Tree

To obtain a visual picture of the hierarchy tree, use the letter 'h' in connection with the `-v` option. For example, using the option `+vbm` for the example in Section 7.5.3 Adding to the Natural Hierarchy you will capture the following hierarchy tree.

```

--Flags
|
|
+--Flags1
| |
| |__Small
|
|__Flags2

```

To get a more compressed tree (vertically) you may follow the 'h' with a '-'. This results in a tree where every other line is removed. For example, if you had used the option `+vh-m` the same tree would appear as:

```

--Flags
|--Flags1
| |__Small
|__Flags2

```

## 7.7 Reference Information

### 7.7.1 Full Source for the Gravitation Example

```

1 //lint -strong(JAc, Meter, Kilogram, Second)
2 //lint -strong(JAc, Area = Meter * Meter)
3 //lint -strong(JAc, Volume = Meter * Meter * Meter)
4 //lint -strong(JAc, Velocity = Meter / Second)
5 //lint -strong(JAc, Acceleration = Meter / (Second * Second))
6 //lint -strong(JAc, Newton = Kilogram * Acceleration)
7 /*lint -strong(JAc, GravitationalConstant =
8         Newton * Area / (Kilogram * Kilogram)
9     )
10 */
11 typedef double Meter, Second, Velocity, Acceleration;
12 typedef double Kilogram, Newton;
13 typedef double Area, Volume;
14 typedef double GravitationalConstant;
15
16 const GravitationalConstant G = 6.67e-11;
17
18 Newton attraction(Kilogram mass1, Kilogram mass2, Meter distance) {
19     return (mass1 * mass2) / (distance * distance);
20 }

```

### 7.7.2 The Strong Type of an Expression

An expression is strongly typed if:

1. It is a strongly typed variable or field.
2. It is the return value of a function whose return type is strong.
3. It is a cast to a strong type.
4. It is one of the type-propagating unary operators, `+`, `-`, `++`, `--`, and `~`, applied to a strongly typed expression.
5. It is the result of dereferencing a pointer to a strong type, or of indexing an array.
6. It is the result of a multiplicative binary operator, `*` or `/`, where at least one operand is dimensional and the operands can be combined through dimensional analysis.
7. It is formed by one of the type-propagating binary operators where both operands are strongly typed expressions with compatible strong types. The type-propagating binary operators are the arithmetic operators, `+`, `-`, `*`, `/`, and `%`, the bitwise operators, `&`, `|`, and `^`, and the conditional operator, `?:`, where the two operands are the true and false arms.
8. It is a shift operator whose left operand is a strong type. The strong type of the shift operator is then the strong type of the left operand.
9. It is a comma operator whose right operand is a strong type. The strong type of the comma operator is then the strong type of the right operand.
10. It is an assignment operator whose left side is a strong type. The strong type of the assignment operator is then the strong type of the left operand.
11. A boolean strong type has been designated and it is a comparison operator, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`, `||`, and `&&`. It will have the strong type of the designated boolean strong type.

### 7.7.3 Canonical Form for Dimensional Strong Types

Every strong type is reduced to a canonical form internally. Dimensional strong types may be specified using any valid C expression containing:

- binary `*` / operators
- identifiers (including the special identifier `1` for numerators)
- balanced parentheses

Output (in messages and the output of `-vh`) will always be presented in the canonical form where all terms are reduced, consecutively multiplied operands are sorted lexicographically, multiplicative expressions as operands to division are parenthesized, a missing numerator is replaced with a `1`, and a denominator of `1` is omitted (and its dividend not parenthesized).

### 7.7.4 Message Numbers

The primary message numbers related to Strong Types are: [18](#), [138](#), [463](#), [632](#), [633](#), [634](#), [635](#), [636](#), [637](#), [638](#), [639](#), [640](#), and [697](#). Setting a strong boolean type will affect the behavior of messages involving boolean contexts that are otherwise unrelated to strong types.

## 8 Value Tracking

### 8.1 Introduction

Most components of PC-Lint Plus analyze a static compile-time view of your code. This is sufficient for most static analysis tasks, but detection of certain runtime problems requires a deeper approach. Value Tracking combines static and dynamic analysis with an in-depth knowledge of C and C++ programming patterns to find potential runtime errors by performing an approximate symbolic interpretation of your code without the exponential slowdown incurred from a blind purely symbolic analysis. For example:

```

1  int g(int x) {
2      x = x - 5;
3      return 100 / x;
4  }
5
6  void f() {
7      int a = 3;
8      a += 2;
9      a = g(a);
10 }
```

Value Tracking will find the division by zero error and explain how it occurs:

```

3 warning 414: possible division by zero
    return 100 / x;
               ^ ~
9 supplemental 894: during specific walk g(5)
    a = g(a);
      ^
2 supplemental 831: assignment yields 0
    x = x - 5;
    ~~~~
2 supplemental 831: operator - yields 0
    x = x - 5;
    ~~~~
1 supplemental 831: argument initialization yields 5
int g(int x) {
    ~~~~^
9 supplemental 831: argument passing yields 5
    a = g(a);
      ^
8 supplemental 831: operator + yields 5
    a += 2;
    ~~~~
```

#### 8.1.1 Anatomy of a Value Tracking Message

The division by zero warning shown in the previous example consists of three parts. The primary message, warning [414](#), describes the problem that Value Tracking has detected. The next message in the message group is supplementary message [894](#), which is used to indicate that the primary message was issued during a specific walk of a function using arguments passed to it in a function call within an earlier walk. Message 894 may occur multiple times in a message group and indicates the values of function arguments on the specific call stack. The repeated instances of message 831 are used to show how the history of a relevant variable led to the occurrence of the situation that was reported as suspicious in the primary message, in

reverse chronological order. The first instance of message 831 shows how `x` came to have the value 0 and the following messages proceed backwards to the value's origin.

## 8.2 Value Inferencing

### 8.2.1 Conditionals

Inside of a conditionally executed region, such as the body of an `if` statement, PC-lint Plus will infer that the condition required for the region to execute must be true when that region begins executing. In the case of an `if` statement, it will further infer that the opposite of such a condition must be true when a matching `else` statement begins executing. Inferred values and modifications made to variables within conditional regions generally do not survive once the conditional scope exits. If the condition can be determined to always be true or false under the circumstances, or one branch of an `if` statement always returns, then changes, inferences, or even a reversed inference may persist beyond the conditionally executed region. Inferencing can be disabled using the `ii` flag, although this is not recommended.

### 8.2.2 Assertions

Values can be inferred from expressions passed as arguments to functions with the *assert* semantic. Inferences derived from assertions work in the same manner as inferences derived from conditionals. The standard `assert` macro is typically redefined as an invocation of `__lint_assert` with the option `++dassert(x)=__lint_assert(!(x))`. `__lint_assert` is a builtin function in PC-lint Plus that takes a single Boolean argument and has the *assert* semantic which will cause PC-lint Plus to assume the condition is true following the invocation. Other user-defined assertion macros can be redefined in the same way. Other functions may be endowed with assertion semantics by copying the semantics of `__lint_assert` using the `-function` option. See Chapter 9 [Semantics](#) for more information. For example:

```

1  #define my_assert(x) __lint_assert(!(x))
2
3  int x;
4
5  int* g() {
6      if (x) return &x;
7      else return nullptr;
8  }
9
10 void f() {
11     int* a = g();
12     my_assert(a);
13     *a = 10; // no warning about potential use of null pointer due to assert
14 }
```

## 8.3 Integer Range Tracking

Rather than tracking only a single value, an integer can be represented by a range of possible values. For example:

```

1  void f(int a) {
2      if (a > 5 && a < 10) {
3          /* a is known to be between six and nine inclusive here */
4      }
5  }
6
7  void g(unsigned a) {
8      if (a > 10) return;
```

```

9      /* a is known to be between zero and ten inclusive here */
10  }
```

You can test these examples using the integrated debugger as explained below in [Section 8.7 Debugger](#).

## 8.4 Terminology

- *General walk* - The initial analysis of a function, independent of the rest of the program. The values of function arguments are completely unknown during a general walk. PC-Lint Plus generally will not report violations relating to these unknown arguments unless it has learned something to constrain their possible values. This helps to avoid false positives. Function calls encountered in the general walk launch specific walks.
- *Specific walk* - A walk of a function launched by a function call during a general walk or an earlier specific walk (up to the depth limit). The arguments passed at the call site are known during this walk.
- *Depth* - The number of nested specific walks at the current point in execution.
- *Pass* - A repeatable event consisting of the processing or reprocessing of each source file. Each file is read from the disk and analyzed again in each pass. Information stored between passes allows some messages to provide more information as the number of passes increases. Default operation involves two passes, one in which analysis local to each module is performed while globally relevant information is collected, and another in which global information collected in the previous pass is acted on for each module (Global Wrap-up). Global Wrap-up only occurs once, in the second pass, if Global Wrap-up is enabled at all. Value Tracking is performed again in each pass.

## 8.5 Value Display Format

### 8.5.1 Integers

The value of an integer will be printed in base ten. If there are a range of possible values, the minimum and maximum values (both inclusive) will be printed, separated by a colon. If the value of an integer is not known but there is reason to believe it may be zero, a zero followed by a question mark will be printed.

### 8.5.2 Pointers in General

If a pointer is null, the string `nullptr` will be printed. If a pointer may possibly be null, a question mark will be printed after the pointer's value. If a pointer is custodial, the value will be followed by a suffix of `C` or `c`, with the lowercase form indicating some degree of uncertainty. A pointer derived from the conversion of an integer literal specifying a fixed constant address will be indicated with a suffix of `F`.

### 8.5.3 Pointers to a Single Datum

A pointer to a unique object not considered to be part of an array or multi-element allocation will be printed as `&(V)` where `V` represents the value of the target object.

### 8.5.4 Pointers to Buffers with Multiple Elements

A pointer into a multi-element allocation will be printed as `[E]@I/S`. `E` represents the size of the allocation in bytes. `I` represents the index into the allocation in bytes. `E` and `I` may be displayed as ranges under the rules specified above for integers. `S` is the size of the element type. The suffix `NUL@Z` represents the belief that a null terminator is present at index `Z`, which may be a range.

### 8.5.5 Objects of Structure or Class Type

Members and base class sub-objects are printed in a comma separated list delimited by curly braces and prefixed with `.` or `:` respectively. For example, given these structures:

```

1  struct X {
2      int a;
3  }
4
5  struct Y : X {
6      int b;
7  };

```

an object of type `Y` may be displayed as `{ :X = { .a = 42 }, .b = 11 }`.

### 8.5.6 File Streams

A *file stream* is the value of a file object represented by a `FILE` pointer such as is returned by `fopen` or `freopen`. PC-lint Plus tracks various attributes of file streams including the file name, the mode in which the file was opened, the orientation of the file, and its current state.

The representation of a `FILE` object looks like:

```
FILE {.filename = "test.txt", .mode = r, .state = opened, .orientation = none}
```

*filename* is the name of the file or `unknown` if the name isn't known. The *mode* represents the mode in which the file was opened, e.g. `rw`, `r+`, etc. The *state* is `opened` if no operations have been performed since the file was opened or `closed` if the file is closed. Otherwise the state represents the last operation performed on the stream with the possible values being:

State Character	Last operation was:
r	read (e.g. <code>fread</code> )
w	write (e.g. <code>fwrite</code> )
u	pushback (e.g. <code>ungetc</code> )
p	file positioning operation (e.g. <code>fseek</code> )
t	a tell operation (e.g. <code>ftell</code> )
f	flush (e.g. <code>fflush</code> )

If the last operation is not known, the possible last operations are reflected in the state followed by a question mark. E.g. `rf?` indicates the last operation on the stream was either a read or a flush.

The orientation represents the file's orientation and is one of `byte`, `wide`, `unknown`, or `none`. The value of `none` means that the file is not oriented because an orientation-inducing operation has not been performed on the file stream since it was opened.

Since file streams represent a `FILE` pointer, the actual file stream will be represented as a *pointer* to a `FILE` object:

```
&(FILE {.filename = unknown, .mode = r, .state = opened, .orientation = none})? C
```

and may include a `?` to indicate the possibility of a null pointer or `C` to indicate the pointer is custodial.

### 8.5.7 Uninitialized Values

Uninitialized values are represented using the string `uninitialized`. A question mark suffix may appear if a value is only possibly uninitialized.

## 8.6 General Usage

Value Tracking is enabled by default. More information can be revealed by increasing the specific walk depth using the `-vt_depth` option. Larger values will increase runtime and (peak) memory usage. A variety of more specific Value Tracking features can be controlled using [9 Semantics](#).

## 8.7 Debugger

A debugger in the spirit of `gdb` is provided to probe the state of the Value Tracking interpreter during execution. This functionality can be accessed using the `+fvd`. The flag can be enabled in a lint comment to avoid triggering the debugger earlier in the program. For example, given `a.cpp`:

```
1 void f(int a) {
2     int b = 5;
3     a = b + 2;
4     int c = a;
5 }
```

running with the debugger enabled will present the following modal interface:

```
@ a.cpp:2
~
# void f(int a) {
--> #     int b = 5;
#     a = b + 2;
#     int c = a;
(vt)
```

which indicates that the debugger has stopped prior to the execution of the line indicated by the arrow in the left margin and listed after the filename in the header. Pressing enter without inputting any text at the `(vt)` prompt will proceed to the next statement.

```
@ a.cpp:3
# void f(int a) {
#     int b = 5;
--> #     a = b + 2;
#     int c = a;
# }
```

Entering `?` will print the current values of all variables in scope:

```
### unknown storage ###
### static storage ###
### dynamic storage ###
### function f parameters ###
a = unknown $1
### compound statement ###
b = 5 $3
```

Each variable belongs to the scope (denoted with triple hash headings) that it appears immediately under. The variable `b` is local to the compound statement of the function body, and has the value 5. The `$3` following the value represents the simulated memory slot in which the value of `b` resides. If the program is stepped to the end of the function, it will stop on the closing brace to provide a final opportunity to issue commands before leaving the scope.

More advanced features are available and listed in the output of the `help` command at the prompt. The debugger is considered experimental and subject to change. Value Tracking debugging is not available when using the Parallel Analysis feature.

## 8.8 Interfunction and Intermodule Value Tracking

Interfunction Value Tracking operates differently depending on whether a given function call is connecting two functions within the same module or two functions within different modules. PC-lint Plus can track values across an arbitrary number of intramodule function calls limited only by the value of the `-vt_depth` option and the amount of time available. The order in which functions are defined within a single module does not influence Value Tracking. While intramodule calls are processed depth-first, intermodule calls are processed breadth-first and the boundary from a given module to any other module can only be crossed in the next pass. The initial processing of each source module constitutes a single pass and an additional pass to utilize the information stored during this pass before the first intermodule results are produced. Another pass will occur by default as part of Global Wrap-up processing. Additional passes can be requested using the `-vt_passes` option.

Calls to library functions will not be walked unless the `flf` flag has been set to 2.

## 8.9 Limitations

### 8.9.1 Initial Values of Static Variables

The initial values of non-`const` static duration variables are not currently considered during Value Tracking. Changes to static variables within a function or call chain *are* tracked. For example:

```

1  int a;
2  int b;
3
4  int f() {
5      return 5 / a;
6  }
7
8  int g() {
9      b = 0;
10     return 5 / b;
11 }
```

In the general walk, PC-lint Plus will report on the division by zero in `g` but will not report on the division by `a` in `f` because while `a` is initialized to 0, there is no information in the function to suggest that `a`, which could have been modified by another function in the program, has any particular value.

### 8.9.2 The Correlated Variables Problem

Correlations between independent variables are not currently tracked. In this example:

```

1  void f(int* p) {
2      int* a = 0;
3      if (p) { a = p; }
4      bool is_null = !p;
5      if (!is_null) {
6          *a = 10;
7      }
8  }
```

PC-lint Plus will report the potential for a null pointer dereference on line 6 while this is technically not possible because the value of `is_null` is correlated with the value of `a`. Nonetheless, reports under these circumstances can be useful because use of this pattern can lead to brittle code. This can be remedied by placing an assertion before line 6, such as `assert(a);`, which will prevent the warning.



### 8.9.3 Terminal Depth Assistance

PC-lint Plus will make an exception to the depth limit for a call to a:

- `constexpr` function
- literal operator
- `const` member function returning `bool`

This helps avoid unexpected results in certain cases where the depth limit would otherwise need to be increased. This is not a substitute for specifying the optimal depth for your desired analysis using the `-vt_depth` option.

## 8.10 Changes from Older Products

- There is no longer a distinction between static and dynamic messages for suppression purposes because the new architecture of PC-lint Plus does not run non-dynamic checks multiple times.
- Value Tracking is now performed depth-first, the same way your program executes on a real machine. Previous products performed a breadth-first search due to the multiple pass architecture used. This implies the removal of `-static_depth`, which is now effectively infinite.
- Integer tracking can now track a range of possible values.
- Floating point values, pointer targets, function pointers, and structure members can now be tracked.
- The reference information format has changed. An example of the new, more detailed reference information is provided in the introduction.
- The "conceivable" severity for conditions dependent on a loop never being entered has been retired.

## 9 Semantics

### 9.1 Function Mimicry (-function)

This section describes how some properties of built-in functions can be transferred to user-defined functions by means of the option `-function`. See also `-printf` and `-scanf`. See also Section 9.2 Semantic Specification to see how to create custom function semantics.

#### 9.1.1 Special Functions

PC-lint Plus is aware of the properties (which we will call semantics) of many standard functions, which we refer to as special functions. A complete list of such functions is shown in Section 9.1.2 Function Listing.

For example, function `fopen()` is recognized as a special function. Its two arguments are checked for the possibility of being the NULL pointer and its return value is considered possibly NULL. Similarly, `fclose` is regarded as a special function whose one argument is also checked for NULL. Thus, the code:

```
if( name ) printf ( "ok\n" );
f = fopen( name, "r" );      // Warning! name may be NULL
fclose ( f );               // Warning! f may be NULL
```

will be greeted with the diagnostics indicated. You may transfer all three semantics of `fopen` to a function of your own, say `myopen`, by using the option

```
-function( fopen, myopen )
```

Then, PC-lint Plus would also check the 1st and 2nd arguments of `myopen` for NULL and assume that the returned pointer could possibly be NULL. In general, the syntax of `-function` is described as follows:

```
-function( Function0, Function1 [, Function2] ... )
```

specifies that *Function1*, *Function2*, etc. are like *Function0* in that they exhibit special properties normally associated with *Function0*.

The arguments to `-function` may be subscripted. For example, if `myopen` were to check its 2nd and 3rd arguments for NULL rather than its 1st and 2nd we could do the following:

```
-function( fopen(1), myopen(2) )
-function( fopen(2), myopen(3) )
```

This would transfer the semantics of NULL checking to the 2nd and 3rd arguments of `myopen`. This could be simplified to

```
-function( fopen(1), myopen(2), myopen(3) )
```

since the property of `fopen(1)` is identical to that of `fopen(2)`. Any previous semantics associated with the 2nd and 3rd arguments to `myopen` would be lost. To transfer the return semantics you may use the option

```
-function( fopen(r), myopen(r) )
```

Some functions have a semantic that is not decomposable to a single argument or return value but is rather a combined property of the entire function. For example

```
char * fread( char *, size_t, size_t, FILE * );
```

has, in addition to the check-for-NULL semantics on its 1st and 4th arguments, and the check-for-negative semantics on the 2nd and 3rd arguments, an additional check to see if the size of argument 2 multiplied by argument 3 exceeds the buffer size given as the 1st argument. This condition is identified as semantic `fread` in Section 9.1.2 Function Listing. Thus

```
char buf[100];
fread( buf, 100, 2, f );      // Warning
```

To transfer this function-wide property to some other function we need to use the 0 (zero) index. Thus

```
-function( fread(0), myread(0) )
```

will transfer just the overflow checking (**fread** as described above) and not the argument checking. That is, of the semantics appearing in Section 9.1.2 [Function Listing](#) for row labeled **fread**, the semantics transferred are only those marked with an asterisk.

As a convenience, the subscript need not be repeated if it is the same as the 1st argument. Thus

```
-function( fread(0), myread )
```

is equivalent to the earlier option.

Just as in the case of **fopen** you may transfer all the properties of **fread** to your own function by not using a subscript as in:

```
function( fread, myread )
```

You may remove any or all of these semantics from a special function by not using a 2nd argument to the **-function** option. Thus

```
-function( fread )
```

will remove all of the semantics of the **fread** function and

```
-function( fread(0) )
```

removes only the special semantics described above.

In summary, an option of the form

```
-function( function(index), ...)
```

copies a single semantic into a destination or destinations. An option of the form

```
-function( function, ... )
```

copies all of a function's semantics.

You may transfer semantics to member functions as well as non-member functions. Thus

```
-function( exit, X::quit )
```

transfers the properties of **exit()** to **X::quit()**. The semantics in this case is simply that the function is not expected to return.

As another example involving member functions consider the following:

```
//lint -function( strlen(1), X::X(1), X::g )
// both X::X() and X::g() should have their 1st
// argument checked for NULL.
//lint +fpn pointer parameters may be NULL

class X {
public:
    char *buf;
    X(char *);
    void g(char *);
};

void f(char *p) {    // p may be NULL because of +fpn
    X x(p);          // Warning 668
    x.g(p);
}
```

In this example, the semantics associated with the 1st argument of `strlen` are transferred to the 1st argument of `X::X` and to the 1st argument of `X::g`. As the example illustrates, when we speak of the  $n$ th argument passed to a member function we are ignoring in our count the implicit argument that is a pointer to the class (this is always checked for NULL).

No distinction is made among overloaded functions. Thus, if `X::X( int * )` is checked for NULL then so is `X::X( char * )`. If there is an `X::X( int )` then its argument is not checked because its argument is not a pointer. If there is an `X::X( int *, char * )` then the 1st argument is checked, but not the 2nd. User-defined semantics can be applied to individual function overloads or function template instantiations, see Section 9.2 [Semantic Specifications](#) for details.

### 9.1.2 Function Listing

<b>DeleteCriticalSection</b> 1p mutex(1) mutex_destroy
<b>EnterCriticalSection</b> 1p mutex(1) mutex_lock
<b>InitializeCriticalSection</b> 1p mutex(1) mutex_is_recursive(1) mutex_initialize
<b>InitializeCriticalSectionAndSpinCount</b> 1p mutex(1) mutex_is_recursive(1) mutex_initialize
<b>IsBadCodePtr</b> dangerous
<b>IsBadHugeReadPtr</b> dangerous
<b>IsBadHugeWritePtr</b> dangerous
<b>IsBadReadPtr</b> dangerous
<b>IsBadStringPtrA</b> dangerous
<b>IsBadStringPtrW</b> dangerous
<b>IsBadWritePtr</b> dangerous
<b>LeaveCriticalSection</b> 1p mutex(1) mutex_unlock
<b>QMutex::QMutex</b> mutex(t) mutex_is_recursive(1) mutex_initialize
<b>QMutex::lock</b> mutex(t) mutex_lock
<b>QMutex::tryLock</b> mutex(t) try_lock_true mutex_lock
<b>QMutex::try_lock</b> mutex(t) try_lock_true mutex_lock
<b>QMutex::try_lock_for</b> mutex(t) try_lock_true mutex_lock
<b>QMutex::try_lock_until</b> mutex(t) try_lock_true mutex_lock

<b>QMutex::unlock</b> mutex(t) mutex_unlock
<b>QMutexLocker::QMutexLocker</b> mutex_locker(t) ip mutex(1) locker_create
<b>QMutexLocker::mutex</b> mutex_locker(t) locker_fetch
<b>QMutexLocker::relock</b> mutex_locker(t) locker_lock
<b>QMutexLocker::unlock</b> mutex_locker(t) locker_unlock
<b>QReadLocker::QReadLocker</b> mutex_is_shared(t) mutex_locker(t) ip mutex(1) locker_create
<b>QReadLocker::readWriteLock</b> mutex_locker(t) locker_fetch
<b>QReadLocker::relock</b> mutex_locker(t) locker_lock_shared
<b>QReadLocker::unlock</b> mutex_locker(t) locker_unlock
<b>QReadWriteLock::QReadWriteLock</b> mutex(t) mutex_is_shared(t) mutex_is_recursive(1) mutex_initialize
<b>QReadWriteLock::lockForRead</b> mutex(t) mutex_lock_shared
<b>QReadWriteLock::lockForWrite</b> mutex(t) mutex_lock
<b>QReadWriteLock::tryLockForRead</b> mutex(t) try_lock_true mutex_lock_shared
<b>QReadWriteLock::tryLockForWrite</b> mutex(t) try_lock_true mutex_lock
<b>QReadWriteLock::unlock</b> mutex(t) mutex_unlock
<b>QRecursiveMutex::QRecursiveMutex</b> mutex(t) mutex_is_recursive(t) mutex_initialize
<b>QRecursiveMutex::lock</b> mutex(t) mutex_lock
<b>QRecursiveMutex::tryLock</b> mutex(t) try_lock_true mutex_lock
<b>QRecursiveMutex::try_lock</b> mutex(t) try_lock_true mutex_lock
<b>QRecursiveMutex::try_lock_for</b> mutex(t) try_lock_true mutex_lock
<b>QRecursiveMutex::try_lock_until</b> mutex(t) try_lock_true mutex_lock
<b>QRecursiveMutex::unlock</b> mutex(t) mutex_unlock
<b>QWriteLocker::QWriteLocker</b> mutex_locker(t) ip mutex(1) locker_create
<b>QWriteLocker::readWriteLock</b> mutex_locker(t) locker_fetch
<b>QWriteLocker::relock</b>

<code>mutex_locker(t) locker_lock</code>
<code>QWriteLocker::unlock</code> <code>mutex_locker(t) locker_unlock</code>
<code>TryEnterCriticalSection</code> <code>1p mutex(1) try_lock_one mutex_lock</code>
<code>_Exit</code> <code>async_signal_safe r_no</code>
<code>--assert</code> <code>*assert</code>
<code>--lint_assert</code> <code>*assert</code>
<code>abort</code> <code>async_signal_safe r_no</code>
<code>acos</code> <code>*dom_1</code>
<code>acosf</code> <code>*dom_1</code>
<code>acosh</code> <code>*dom_lt1</code>
<code>acoshf</code> <code>*dom_lt1</code>
<code>acoshl</code> <code>*dom_lt1</code>
<code>acosl</code> <code>*dom_1</code>
<code>asctime</code> <code>1p</code>
<code>asctime_s</code> <code>1p chneg(2) 3p</code>
<code>asin</code> <code>*dom_1</code>
<code>asinf</code> <code>*dom_1</code>
<code>asinh</code> <code>*dom_1</code>
<code>at_quick_exit</code> <code>1p</code>
<code>atanh</code> <code>*dom_1</code>
<code>atanhf</code> <code>*dom_1</code>
<code>atanhl</code> <code>*dom_1</code>
<code>atexit</code> <code>1p</code>
<code>atof</code> <code>1p</code>
<code>atoi</code> <code>1p</code>

<code>atol</code>
1p
<code>atoll</code>
1p
<code>atomic_compare_exchange_strong</code>
1p thread_atomic(1) 2p
<code>atomic_compare_exchange_strong_explicit</code>
1p thread_atomic(1) 2p
<code>atomic_compare_exchange_weak</code>
1p thread_atomic(1) 2p
<code>atomic_compare_exchange_weak_explicit</code>
1p thread_atomic(1) 2p
<code>atomic_exchange</code>
1p thread_atomic(1)
<code>atomic_exchange_explicit</code>
1p thread_atomic(1)
<code>atomic_fetch_add</code>
1p thread_atomic(1)
<code>atomic_fetch_add_explicit</code>
1p thread_atomic(1)
<code>atomic_fetch_and</code>
1p thread_atomic(1)
<code>atomic_fetch_and_explicit</code>
1p thread_atomic(1)
<code>atomic_fetch_or</code>
1p thread_atomic(1)
<code>atomic_fetch_or_explicit</code>
1p thread_atomic(1)
<code>atomic_fetch_sub</code>
1p thread_atomic(1)
<code>atomic_fetch_sub_explicit</code>
1p thread_atomic(1)
<code>atomic_fetch_xor</code>
1p thread_atomic(1)
<code>atomic_fetch_xor_explicit</code>
1p thread_atomic(1)
<code>atomic_flag_clear</code>
1p thread_atomic(1)
<code>atomic_flag_clear_explicit</code>
1p thread_atomic(1)
<code>atomic_flag_test_and_set</code>
1p thread_atomic(1)
<code>atomic_flag_test_and_set_explicit</code>
1p thread_atomic(1)
<code>atomic_init</code>
1p thread_non_atomic(1)
<code>atomic_load</code>
1p thread_atomic(1)
<code>atomic_load_explicit</code>

<b>1p thread_atomic(1)</b>
<b>atomic_store</b> 1p thread_atomic(1)
<b>atomic_store_explicit</b> 1p thread_atomic(1)
<b>bsearch</b> 1p 2p chneg(3) chneg(4) 5p r_null
<b>bsearch_s</b> chneg(3) chneg(4)
<b>call_once</b> 1p 2p
<b>calloc</b> chneg(1) chneg(2) r_null *calloc
<b>clearerr</b> 1p stream_placid(1)
<b>clearerr_unlocked</b> 1p stream_placid(1)
<b>cnd_broadcast</b> 1p
<b>cnd_destroy</b> 1p
<b>cnd_init</b> 1p
<b>cnd_signal</b> 1p
<b>cnd_timedwait</b> 1p 2p mutex(2) mutex_must_be_locked(2) 3p mutex_validate
<b>cnd_wait</b> 1p 2p mutex(2) mutex_must_be_locked(2) mutex_validate
<b>ctime</b> 1p
<b>ctime_s</b> 1p chneg(2) 3p
<b>exit</b> r_no
<b>fclose</b> 1p custodial(1) stream_close(1)
<b>feof</b> 1p stream_placid(1)
<b>feof_unlocked</b> 1p stream_placid(1)
<b>ferror</b> 1p stream_placid(1)
<b>ferror_unlocked</b> 1p stream_placid(1)
<b>fflush</b> stream_flush(1)
<b>fflush_unlocked</b> 1p stream_flush(1)



<b>fgetc</b> 1p stream_read(1) stream_byte(1)
<b>fgetc_unlocked</b> 1p stream_read(1) stream_byte(1)
<b>fgetpos</b> 1p 2p
<b>fgets</b> 1p chneg(2) 3p stream_read(3) stream_byte(3) r_null *fgets
<b>fgets_unlocked</b> 1p chneg(2) 3p stream_read(3) stream_byte(3) r_null *fgets
<b>fgetwc</b> 1p stream_read(1) stream_wide(1)
<b>fgetwc_unlocked</b> 1p stream_read(1) stream_wide(1)
<b>fgetws</b> 1p chneg(2) 3p stream_read(3) stream_wide(3) r_null *fgets
<b>fgetws_unlocked</b> 1p chneg(2) 3p stream_read(3) stream_wide(3) r_null *fgets
<b>fileno</b> 1p stream_placid(1)
<b>fileno_unlocked</b> 1p stream_placid(1)
<b>flockfile</b> 1p stream_placid(1)
<b>fopen</b> 1p 2p r_null *fopen
<b>fopen_s</b> 1p 2p 3p
<b>fprintf</b> 1p stream_write(1) stream_byte(1) 2p printf(2)
<b>fprintf_s</b> 1p stream_write(1) stream_byte(1) 2p printf(2)
<b>fputc</b> 2p stream_write(2) stream_byte(2)
<b>fputc_unlocked</b> 2p stream_write(2) stream_byte(2)
<b>fputs</b> 1p 2p stream_write(2) stream_byte(2)
<b>fputs_unlocked</b> 1p 2p stream_write(2) stream_byte(2)
<b>fputwc</b> 2p stream_write(2) stream_wide(2)
<b>fputwc_unlocked</b> 2p stream_write(2) stream_wide(2)
<b>fputws</b> 1p 2p stream_write(2) stream_wide(2)
<b>fputws_unlocked</b> 1p 2p stream_write(2) stream_wide(2)
<b>fread</b>

1p chneg(2) chneg(3) 4p stream_read(4) stream_byte(4) <i>*fread</i>
<b>fread_unlocked</b> 1p chneg(2) chneg(3) 4p stream_read(4) stream_byte(4) <i>*fread</i>
<b>free</b> <i>*free</i>
<b>freopen</b> 2p 3p r_null <i>*freopen</i>
<b>freopen_s</b> 1p 3p 4p
<b>frexp</b> 2p
<b>frexpf</b> 2p
<b>fscanf</b> 1p stream_read(1) stream_byte(1) 2p scanf(2)
<b>fscanf_s</b> 1p stream_read(1) stream_byte(1) 2p scanf(2)
<b>fseek</b> 1p stream_pos(1)
<b>fsetpos</b> 1p stream_pos(1) 2p
<b>ftell</b> 1p
<b>ftrylockfile</b> 1p stream_placid(1)
<b>funlockfile</b> 1p stream_placid(1)
<b>fwide</b> 1p stream_placid(1)
<b>fwprintf</b> 1p stream_write(1) stream_wide(1) 2p printf(2)
<b>fwprintf_s</b> 1p stream_write(1) stream_wide(1) 2p printf(2)
<b>fwrite</b> 1p chneg(2) chneg(3) 4p stream_write(4) stream_byte(4) <i>*fwrite</i>
<b>fwrite_unlocked</b> 1p chneg(2) chneg(3) 4p stream_write(4) stream_byte(4) <i>*fwrite</i>
<b>fwscanf</b> 1p stream_read(1) stream_wide(1) 2p scanf(2)
<b>fwscanf_s</b> 1p stream_read(1) stream_wide(1) 2p scanf(2)
<b>getc</b> 1p stream_read(1) stream_byte(1)
<b>getc_unlocked</b> 1p stream_read(1) stream_byte(1)
<b>getenv</b> 1p
<b>getenv_s</b> 1p chneg(3)

<b>gets</b> 1p dangerous r_null
<b>gets_s</b> 1p chneg(2)
<b>getwc</b> 1p stream_read(1) stream_wide(1)
<b>getwc_unlocked</b> 1p stream_read(1) stream_wide(1)
<b>gmtime</b> 1p r_null
<b>gmtime_s</b> 1p 2p
<b>localtime</b> 1p r_null
<b>localtime_s</b> 1p 2p
<b>log</b> <i>*dom_lt0</i>
<b>log10</b> <i>*dom_lt0</i>
<b>log10f</b> <i>*dom_lt0</i>
<b>log10l</b> <i>*dom_lt0</i>
<b>log1p</b> <i>*dom_ltn1</i>
<b>log1pf</b> <i>*dom_ltn1</i>
<b>log1pl</b> <i>*dom_ltn1</i>
<b>log2</b> <i>*dom_lt0</i>
<b>log2f</b> <i>*dom_lt0</i>
<b>log2l</b> <i>*dom_lt0</i>
<b>logf</b> <i>*dom_lt0</i>
<b>logl</b> <i>*dom_lt0</i>
<b>longjmp</b> 1p r_no
<b>malloc</b> chneg(1) r_null <i>*malloc</i>
<b>mbsrtowcs_s</b> 1p chneg(3) 4p chneg(5) 6p
<b>mbstowcs</b> 1p 2p chneg(3) <i>*mbstowcs</i>
<b>mbstowcs_s</b>

1p chneg(3) 4p chneg(5)
<b>memchr</b> 1p pod(1) chneg(3) r_null *memchr
<b>memcmp</b> 1p pod(1) 2p pod(2) chneg(3) *memcmp
<b>memcpy</b> 1p pod(1) 2p pod(2) chneg(3) *memcpy
<b>memcpy_s</b> 1p pod(1) chneg(2) 3p pod(3) chneg(4)
<b>memmove</b> 1p pod(1) 2p pod(2) chneg(3) *memmove
<b>memmove_s</b> 1p pod(1) chneg(2) 3p pod(3) chneg(4)
<b>memset</b> 1p pod(1) chneg(3) *memset
<b>memset_s</b> 1p pod(1) chneg(2) chneg(3)
<b>mktime</b> 1p inout(1)
<b>modf</b> 2p
<b>modff</b> 2p
<b>modfl</b> 2p
<b>mtx_destroy</b> 1p mutex(1) mutex_destroy
<b>mtx_init</b> 1p mutex(1) mutex_is_recursive(2) mutex_initialize_std_c
<b>mtx_lock</b> 1p mutex(1) try_lock_std_c mutex_lock
<b>mtx_timedlock</b> 1p mutex(1) 2p try_lock_std_c mutex_lock
<b>mtx_trylock</b> 1p mutex(1) try_lock_std_c mutex_lock
<b>mtx_unlock</b> 1p mutex(1) mutex_unlock
<b>perror</b> 1p
<b>printf</b> 1p printf(1)
<b>printf_s</b> 1p printf(1)
<b>pthread_cond_timedwait</b> 1p 2p mutex(2) 3p mutex_validate
<b>pthread_cond_wait</b> 1p 2p mutex(2) mutex_validate
<b>pthread_create</b> 1p 3p thread_create(3) thread_args(4) mutex_ignore

<b>pthread_mutex_consistent</b> 1p mutex(1) mutex_validate
<b>pthread_mutex_destroy</b> 1p mutex(1) mutex_destroy
<b>pthread_mutex_getprioceiling</b> 1p mutex(1) 2p mutex_validate
<b>pthread_mutex_init</b> 1p mutex(1) mutex_attribute(2) mutex_initialize
<b>pthread_mutex_lock</b> 1p mutex(1) try_lock_zero mutex_lock
<b>pthread_mutex_setprioceiling</b> 1p mutex(1) 3p mutex_validate
<b>pthread_mutex_timedlock</b> 1p mutex(1) 2p try_lock_zero mutex_lock
<b>pthread_mutex_trylock</b> 1p mutex(1) try_lock_zero mutex_lock
<b>pthread_mutex_unlock</b> 1p mutex(1) mutex_unlock
<b>pthread_mutexattr_destroy</b> 1p mutex_attribute(1) mutex_attribute_destroy
<b>pthread_mutexattr_getprioceiling</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_mutexattr_getprotocol</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_mutexattr_getpshared</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_mutexattr_getrobust</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_mutexattr_gettype</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_mutexattr_init</b> 1p mutex_attribute(1) mutex_attribute_initialize
<b>pthread_mutexattr_setprioceiling</b> 1p mutex_attribute(1) mutex_validate
<b>pthread_mutexattr_setprotocol</b> 1p mutex_attribute(1) mutex_validate
<b>pthread_mutexattr_setpshared</b> 1p mutex_attribute(1) mutex_validate
<b>pthread_mutexattr_setrobust</b> 1p mutex_attribute(1) mutex_validate
<b>pthread_mutexattr_settype</b> 1p mutex_attribute(1) mutex_is_recursive(2) mutex_attribute_set
<b>pthread_rwlock_destroy</b> 1p mutex(1) mutex_destroy
<b>pthread_rwlock_init</b> 1p mutex(1) mutex_is_shared(1) mutex_attribute(2) mutex_initialize
<b>pthread_rwlock_rdlock</b> 1p mutex(1) try_lock_zero mutex_lock_shared
<b>pthread_rwlock_tryrdlock</b>

<b>1p mutex(1) try_lock_zero mutex_lock_shared</b>
<b>pthread_rwlock_trywrlock</b> 1p mutex(1) try_lock_zero mutex_lock
<b>pthread_rwlock_unlock</b> 1p mutex(1) mutex_unlock
<b>pthread_rwlock_wrlock</b> 1p mutex(1) try_lock_zero mutex_lock
<b>pthread_rwlockattr_destroy</b> 1p mutex_attribute(1) mutex_attribute_destroy
<b>pthread_rwlockattr_getpshared</b> 1p mutex_attribute(1) 2p mutex_validate
<b>pthread_rwlockattr_init</b> 1p mutex_attribute(1) mutex_is_recursive(1) mutex_is_shared(1) mutex_attribute_initialize
<b>pthread_rwlockattr_setpshared</b> 1p mutex_attribute(1) mutex_validate
<b>putc</b> 2p stream_write(2) stream_byte(2)
<b>putc_unlocked</b> 2p stream_write(2) stream_byte(2)
<b>puts</b> 1p
<b>putwc</b> 2p stream_write(2) stream_wide(2)
<b>putwc_unlocked</b> 2p stream_write(2) stream_wide(2)
<b>qsort</b> 1p inout(1) chneg(2) chneg(3) 4p
<b>qsort_s</b> inout(1) chneg(2) chneg(3)
<b>quick_exit</b> async_signal_safe r_no
<b>realloc</b> r_null *realloc
<b>remove</b> 1p
<b>rename</b> 1p 2p
<b>rewind</b> 1p stream_pos(1)
<b>scanf</b> 1p scanf(1)
<b>scanf_s</b> 1p scanf(1)
<b>setbuf</b> 1p stream_placid(1)
<b>setbuffer</b> 1p stream_placid(1)
<b>setlinebuf</b> 1p stream_placid(1)

<b>setvbuf</b> 1p stream_placid(1)
<b>sigaction</b> signal_register(2)
<b>signal</b> signal_register(2)
<b>snprintf</b> chneg(2) 3p printf(3) <i>*sprintf</i>
<b>snprintf_s</b> 1p chneg(2) 3p printf(3) <i>*sprintf</i>
<b>snwprintf_s</b> 1p chneg(2) 3p printf(3)
<b>sprintf</b> 1p 2p printf(2) <i>*sprintf</i>
<b>sprintf_s</b> 1p chneg(2) 3p printf(3) <i>*sprintf</i>
<b>sqrt</b> <i>*dom_lt0</i>
<b>sqrtf</b> <i>*dom_lt0</i>
<b>sqrtl</b> <i>*dom_lt0</i>
<b>sscanf</b> 1p 2p scanf(2)
<b>sscanf_s</b> 1p 2p scanf(2)
<b>std::async</b> async
<b>std::atomic::atomic</b> thread_non_atomic(t)
<b>std::atomic_flag::atomic_flag</b> thread_non_atomic(t)
<b>std::condition_variable::condition_variable</b> mutex_ignore
<b>std::condition_variable::native_handle</b> mutex_ignore
<b>std::condition_variable::notify_all</b> thread_immune(t) mutex_ignore
<b>std::condition_variable::notify_one</b> thread_immune(t) mutex_ignore
<b>std::condition_variable::wait</b> thread_immune(t) 1p mutex_locker(1) mutex_must_be_locked(1) mutex_validate
<b>std::condition_variable::wait_for</b> thread_immune(t) 1p mutex_locker(1) mutex_must_be_locked(1) mutex_validate
<b>std::condition_variable::wait_until</b> thread_immune(t) 1p mutex_locker(1) mutex_must_be_locked(1) mutex_validate
<b>std::condition_variable_any::condition_variable_any</b> mutex_ignore
<b>std::condition_variable_any::native_handle</b>

<code>mutex_ignore</code>
<code>std::condition_variable_any::notify_all</code> <code>thread_immune(t) mutex_ignore</code>
<code>std::condition_variable_any::notify_one</code> <code>thread_immune(t) mutex_ignore</code>
<code>std::condition_variable_any::wait</code> <code>thread_immune(t) 1p mutex(1) mutex_must_be_locked(1) mutex_validate</code>
<code>std::condition_variable_any::wait_for</code> <code>thread_immune(t) 1p mutex(1) mutex_must_be_locked(1) mutex_validate</code>
<code>std::condition_variable_any::wait_until</code> <code>thread_immune(t) 1p mutex(1) mutex_must_be_locked(1) mutex_validate</code>
<code>std::lock</code> <code>mutex_remaining(1) mutex_lock</code>
<code>std::lock_guard::lock_guard</code> <code>mutex_locker(t) 1p mutex(1) mutex_tag_adapt(2) locker_create</code>
<code>std::mutex::lock</code> <code>mutex(t) mutex_lock</code>
<code>std::mutex::mutex</code> <code>mutex(t) mutex_attribute(t) mutex_initialize</code>
<code>std::mutex::try_lock</code> <code>mutex(t) try_lock_true mutex_lock</code>
<code>std::mutex::unlock</code> <code>mutex(t) mutex_unlock</code>
<code>std::recursive_mutex::lock</code> <code>mutex(t) mutex_lock</code>
<code>std::recursive_mutex::recursive_mutex</code> <code>mutex(t) mutex_is_recursive(t) mutex_initialize</code>
<code>std::recursive_mutex::try_lock</code> <code>mutex(t) try_lock_true mutex_lock</code>
<code>std::recursive_mutex::unlock</code> <code>mutex(t) mutex_unlock</code>
<code>std::recursive_timed_mutex::lock</code> <code>mutex(t) mutex_lock</code>
<code>std::recursive_timed_mutex::recursive_timed_mutex</code> <code>mutex(t) mutex_is_recursive(t) mutex_initialize</code>
<code>std::recursive_timed_mutex::try_lock</code> <code>mutex(t) try_lock_true mutex_lock</code>
<code>std::recursive_timed_mutex::try_lock_for</code> <code>mutex(t) try_lock_true mutex_lock</code>
<code>std::recursive_timed_mutex::try_lock_until</code> <code>mutex(t) try_lock_true mutex_lock</code>
<code>std::recursive_timed_mutex::unlock</code> <code>mutex(t) mutex_unlock</code>
<code>std::scoped_lock::scoped_lock</code> <code>mutex_locker(t) mutex_remaining(1) mutex_tag_adapt(1) locker_create</code>
<code>std::shared_lock::lock</code> <code>mutex_locker(t) locker_lock_shared</code>
<code>std::shared_lock::mutex</code> <code>mutex_locker(t) locker_fetch</code>



<b>std::shared_lock::owns_lock</b> mutex_locker(t) try_lock_true locker_owns
<b>std::shared_lock::release</b> mutex_locker(t) locker_release
<b>std::shared_lock::shared_lock</b> mutex_is_shared(t) mutex_locker(t) 1p mutex(1) mutex_tag_adopt(2) mutex_tag_defer(2) mutex_tag_try_to_lock(2) locker_create
<b>std::shared_lock::swap</b> mutex_locker(t) 1p mutex_locker(1) locker_swap
<b>std::shared_lock::try_lock</b> mutex_locker(t) try_lock_true locker_lock_shared
<b>std::shared_lock::try_lock_for</b> mutex_locker(t) try_lock_true locker_lock_shared
<b>std::shared_lock::try_lock_until</b> mutex_locker(t) try_lock_true locker_lock_shared
<b>std::shared_lock::unlock</b> mutex_locker(t) locker_unlock
<b>std::shared_mutex::lock</b> mutex(t) mutex_lock
<b>std::shared_mutex::lock_shared</b> mutex(t) mutex_lock_shared
<b>std::shared_mutex::shared_mutex</b> mutex(t) mutex_is_shared(t) mutex_initialize
<b>std::shared_mutex::try_lock</b> mutex(t) try_lock_true mutex_lock
<b>std::shared_mutex::try_lock_shared</b> mutex(t) try_lock_true mutex_lock
<b>std::shared_mutex::unlock</b> mutex(t) mutex_unlock
<b>std::shared_mutex::unlock_shared</b> mutex(t) mutex_unlock_shared
<b>std::shared_timed_mutex::lock</b> mutex(t) mutex_lock
<b>std::shared_timed_mutex::lock_shared</b> mutex(t) mutex_lock_shared
<b>std::shared_timed_mutex::shared_timed_mutex</b> mutex(t) mutex_is_shared(t) mutex_initialize
<b>std::shared_timed_mutex::try_lock</b> mutex(t) try_lock_true mutex_lock
<b>std::shared_timed_mutex::try_lock_for</b> mutex(t) try_lock_true mutex_lock
<b>std::shared_timed_mutex::try_lock_shared</b> mutex(t) try_lock_true mutex_lock_shared
<b>std::shared_timed_mutex::try_lock_shared_for</b> mutex(t) try_lock_true mutex_lock_shared
<b>std::shared_timed_mutex::try_lock_shared_until</b> mutex(t) try_lock_true mutex_lock_shared
<b>std::shared_timed_mutex::try_lock_until</b> mutex(t) try_lock_true mutex_lock

<b>std::shared_timed_mutex::unlock</b> mutex(t) mutex_unlock
<b>std::shared_timed_mutex::unlock_shared</b> mutex(t) mutex_unlock_shared
<b>std::thread::thread</b> 1p thread_create(1) thread_args(2) mutex_ignore
<b>std::timed_mutex::lock</b> mutex(t) mutex_lock
<b>std::timed_mutex::timed_mutex</b> mutex(t) mutex_attribute(t) mutex_initialize
<b>std::timed_mutex::try_lock</b> mutex(t) try_lock_true mutex_lock
<b>std::timed_mutex::try_lock_for</b> mutex(t) try_lock_true mutex_lock
<b>std::timed_mutex::try_lock_until</b> mutex(t) try_lock_true mutex_lock
<b>std::timed_mutex::unlock</b> mutex(t) mutex_unlock
<b>std::try_lock</b> mutex_remaining(1) try_lock_neg_1 mutex_lock
<b>std::uncaught_exception</b> dangerous
<b>std::unique_lock::lock</b> mutex_locker(t) locker_lock
<b>std::unique_lock::mutex</b> mutex_locker(t) locker_fetch
<b>std::unique_lock::owns_lock</b> mutex_locker(t) try_lock_true locker_owns
<b>std::unique_lock::release</b> mutex_locker(t) locker_release
<b>std::unique_lock::swap</b> mutex_locker(t) 1p mutex_locker(1) locker_swap
<b>std::unique_lock::try_lock</b> mutex_locker(t) try_lock_true locker_lock
<b>std::unique_lock::try_lock_for</b> mutex_locker(t) try_lock_true locker_lock
<b>std::unique_lock::try_lock_until</b> mutex_locker(t) try_lock_true locker_lock
<b>std::unique_lock::unique_lock</b> mutex_locker(t) 1p mutex(1) mutex_tag_adopt(2) mutex_tag_defer(2) mutex_tag_try_to_lock(2) locker_create
<b>std::unique_lock::unlock</b> mutex_locker(t) locker_unlock
<b>strcat</b> 1p inout(1) 2p *strcat
<b>strcat_s</b> 1p inout(1) chneg(2) 3p
<b>strchr</b> 1p type(1) r_null

<b>strcmp</b> 1p 2p
<b>strcoll</b> 1p 2p
<b>strcpy</b> 1p 2p <i>*strcpy</i>
<b>strcpy_s</b> 1p chneg(2) 3p
<b>strcspn</b> 1p 2p
<b>strerror_s</b> 1p chneg(2)
<b>strftime</b> 1p chneg(2) 3p 4p
<b>strlen</b> 1p <i>*strlen</i>
<b>strncat</b> 1p inout(1) 2p chneg(3) <i>*strncat</i>
<b>strncat_s</b> 1p inout(1) chneg(2) 3p chneg(4)
<b>strncmp</b> 1p 2p chneg(3)
<b>strncpy</b> 1p 2p chneg(3) <i>*strncpy</i>
<b>strncpy_s</b> 1p chneg(2) 3p chneg(4)
<b>strpbrk</b> 1p type(1) 2p r_null
<b>strrchr</b> 1p type(1) r_null
<b>strspn</b> 1p 2p
<b>strstr</b> 1p type(1) 2p r_null
<b>strtod</b> 1p
<b>strtodf</b> 1p
<b>strtok</b> inout(1) 2p r_null
<b>strtok_s</b> inout(1) 2p 3p 4p
<b>strtol</b> 1p
<b>strtold</b> 1p
<b>strtoll</b> 1p
<b>strtoul</b>

1p
<b>strtoull</b>
1p
<b>strxfrm</b>
2p chneg(3) <i>*strxfrm</i>
<b>swprintf</b>
1p chneg(2) 3p printf(3)
<b>swprintf_s</b>
1p chneg(2) 3p printf(3)
<b>swscanf</b>
1p 2p scanf(2)
<b>swscanf_s</b>
1p 2p scanf(2)
<b>thrd_create</b>
1p 2p thread_create(2) thread_args(3)
<b>thrd_sleep</b>
1p
<b>timespec_get</b>
1p
<b>tmpfile</b>
r_null
<b>tmpfile_s</b>
1p
<b>tmpnam_s</b>
1p chneg(2)
<b>tss_create</b>
1p
<b>ungetc</b>
2p stream_push(2) stream_byte(2)
<b>ungetwc</b>
2p stream_push(2) stream_wide(2)
<b>vfprintf</b>
1p stream_write(1) stream_byte(1) 2p 3p printf(2)
<b>vfprintf_s</b>
1p stream_write(1) stream_byte(1) 2p 3p printf(2)
<b>vfscanf</b>
1p stream_read(1) stream_byte(1) 2p 3p scanf(2)
<b>vfscanf_s</b>
1p stream_read(1) stream_byte(1) 2p 3p scanf(2)
<b>vfwprintf</b>
1p stream_write(1) stream_wide(1) 2p 3p printf(2)
<b>vfwprintf_s</b>
1p stream_write(1) stream_wide(1) 2p 3p printf(2)
<b>vfwscanf</b>
1p stream_read(1) stream_wide(1) 2p 3p scanf(2)
<b>vfwscanf_s</b>
1p stream_read(1) stream_wide(1) 2p 3p scanf(2)
<b>vprintf</b>
1p 2p printf(1)

<b>vprintf_s</b> 1p 2p printf(1)
<b>vscanf</b> 1p 2p scanf(1)
<b>vscanf_s</b> 1p 2p scanf(1)
<b>vsprintf</b> chneg(2) 3p 4p printf(3)
<b>vsprintf_s</b> 1p chneg(2) 3p 4p printf(3)
<b>vsnwprintf_s</b> 1p chneg(2) 3p 4p printf(3)
<b>vsprintf</b> 1p 2p 3p printf(2)
<b>vsprintf_s</b> 1p chneg(2) 3p 4p printf(3)
<b>vsscanf</b> 1p 2p 3p scanf(2)
<b>vsscanf_s</b> 1p 2p 3p scanf(2)
<b>vswprintf_s</b> 1p chneg(2) 3p 4p printf(3)
<b>vswscanf_s</b> 1p 2p 3p scanf(2)
<b>vwprintf_s</b> 1p 2p printf(1)
<b>vwscanf_s</b> 1p 2p scanf(1)
<b>wcrtomb_s</b> 1p chneg(3) 5p
<b>wscat_s</b> 1p chneg(2) 3p
<b>wscpy_s</b> 1p chneg(2) 3p
<b>wcsncat_s</b> 1p chneg(2) 3p chneg(4)
<b>wcsncpy_s</b> 1p chneg(2) 3p chneg(4)
<b>wcsrtombs_s</b> 1p chneg(3) 4p chneg(5) 6p
<b>wcstok_s</b> 2p 3p 4p
<b>wcstombs</b> 1p 2p chneg(3) *wcstombs
<b>wctomb</b> 1p
<b>wmemcpy_s</b> 1p chneg(2) 3p chneg(4)
<b>wmemmove_s</b>

1p chneg(2) 3p chneg(4)
<b>wprintf</b> 1p printf(1)
<b>wprintf_s</b> 1p printf(1)
<b>wscanf</b> 1p 2p scanf(2)
<b>wscanf_s</b> 1p scanf(1)

## Semantics

<b>assert</b>	The function argument can be assumed to be true (non-zero).
<b>calloc</b>	The length of the returned buffer is the product of the first and second arguments or the returned pointer is NULL.
<b>dom_1</b>	The specified argument(s) must be in the range of [-1, 1] as a value outside this range is not defined for this function and may result in a domain error; violations will be diagnosed with message <a href="#">2423/2623</a> .
<b>dom_lt1</b>	The specified argument(s) must not be less than 1 as such a value is not defined for this function and may result in a domain error; violations will be diagnosed with message <a href="#">2423/2623</a> .
<b>dom_ltn1</b>	The specified argument(s) must not be less than -1 as such a value is not defined for this function and may result in a domain error; violations will be diagnosed with message <a href="#">2423/2623</a> .
<b>dom_lt0</b>	The specified argument(s) must not be less than 0 as such a value is not defined for this function and may result in a domain error; violations will be diagnosed with message <a href="#">2423/2623</a> .
<b>exit</b>	The function never returns.
<b>fclose</b>	Pointer argument 1 is regarded as being uninitialized after the function returns.
<b>fgets</b>	Integer argument 2 should not exceed the size of the buffer pointed to by argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<b>fread</b>	The product of the integer arguments 2 and 3 should not exceed the size of buffer argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<b>free</b>	Pointer argument 1 is regarded as being uninitialized after the function returns and the pointed to memory is marked as having been freed (attempting to free the same memory a second time will be diagnosed by message <a href="#">449</a> ). Additionally, if the memory pointed to by argument 1 was derived from an allocation source not appropriate for deallocation via the <b>free</b> function, this will be diagnosed via message <a href="#">424</a> .
<b>fwrite</b>	The product of the integer arguments 2 and 3 should not exceed the size of buffer argument 1; violations will be diagnosed with message <a href="#">420/670</a> (access beyond end of array).
<b>malloc</b>	The length of the buffer returned is the value of integer argument 1 or the returned pointer is NULL.
<b>memchr</b>	Integer argument 3 should not be larger than the size of the buffer pointed to by argument 1; violations will be diagnosed with message <a href="#">420/670</a> (access beyond end of array).
<b>memcmp</b>	Integer argument 3 should not be larger than the size of the buffer pointed to by either argument 1 or argument 2; violations will be diagnosed with message <a href="#">420/670</a> (access beyond end of array).

<code>memcpy</code>	Integer argument 3 should not be larger than the size of the buffer pointed to by either argument 1 or argument 2; a value that exceeds argument 1 will be diagnosed with message <a href="#">419/669</a> (data overrun), a value that exceeds argument 2 will be diagnosed with message <a href="#">420/670</a> (access beyond end of array).
<code>memset</code>	Integer argument 3 should not be larger than the size of the buffer pointed to by argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<code>sprintf</code>	Message <a href="#">464</a> will be issued if the call to this <code>sprintf</code> -like function will result in the destination string being written onto itself.
<code>realloc_1</code>	Pointer argument 1 is regarded as being possibly uninitialized after the function returns.
<code>realloc</code>	The length of the buffer returned is the value of integer argument 2 or the returned pointer is NULL.
<code>strcat</code>	The size of buffer argument 2 should not be larger than the size of buffer argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<code>strcpy</code>	The size of buffer argument 2 should not be larger than the size of buffer argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<code>strncat</code>	Integer argument 3 should not be larger than the size of buffer argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).
<code>strncpy</code>	Integer argument 3 should not be larger than the size of buffer argument 1; violations will be diagnosed with message <a href="#">419/669</a> (data overrun).

### 9.1.3 Other names with special behavior that cannot be mimicked

A function named `main` at global scope is treated as a special case in certain contexts where the usual behavior would be inappropriate. For example, message [714](#) will not be issued as `main` is typically not explicitly called (and such a call is prohibited in C++ [4, `basic.start.main`]).

The functions `rand`, `srand`, `random`, `srandom`, and `time` at global scope or within namespace `std` are specifically considered by name for the purposes of messages [2460](#), [2461](#), [2760](#), and [2960](#).

A call to the function `std::addressof` is specifically considered by name to constitute taking the address of a sub-expression in certain contexts where unary `&` would otherwise be expected.

## 9.2 Semantic Specifications (`-sem`)

The `-sem()` option allows the user to endow his functions with user-defined semantics. This may be considered an extension of the `-function()` option (See Section [9.1 Function Mimicry \(-function\)](#)). Recall that with the `-function()` option the user may copy the semantics of a built-in function to any other function but new semantics cannot be created.

With the `-sem` option, entirely new checks can be created; integral and pointer arguments can be checked in combination with each other using usual C operators and syntax. Also, you can specify some constraints upon the return value.

The format of the `-sem()` option is:

```
-sem( function[,sem] ...)
```

This associates the semantics `sem ...` with the named function `function`. The semantics `sem` are defined below. If no `sem` is given, i.e. if only `function` is given, the option is taken as a request to remove semantics from the named function. Once semantics have been given to a named function, the `-function()` option may be used to copy the semantics in whole or in part to other functions.

### 9.2.1 Possible Semantics

*sem* may be one of:

**r\_null** the function may return the null pointer.

This information is used in subsequent value tracking. For example:

```
/*lint -sem( f, r_null ) */
char *f();
char *p = f();
*p = 0; /* warning, p may be null */
```

This is the same semantic that is employed for the builtin function semantics such as **bsearch**, **calloc**, and **fgets** in Section 9.1.2 [Function Listing](#), and it is considered a Return semantic. See Section 9.1 [Function Mimicry \(-function\)](#) for the definition of Return semantic. A more flexible way to provide Return semantics is given below under expressions (*exp*).

**r\_no** the function does not return.

Code following such a function is considered unreachable. This semantic is identical to the semantic used for the **exit()** function as shown in Section 9.1.2 [Function Listing](#). This also is considered a Return semantic.

**ip** (e.g. 3p) the *i*th argument should be checked for null.

If the *i*th argument could possibly be null this will be reported. For example:

```
/*lint -sem( g, 1p ) warn if g() is passed a NULL */
/*lint -sem( f, r_null ) f() may return NULL */
char *f();
void g(char *);
g( f() ); /* warning, g is passed a possible null */
```

#### initializer

Some member functions are used to initialize members. They may be called from constructors or called directly when the programmer wants to reset the state of a class to what it would have been immediately after construction. In most cases, PC-lint Plus can automatically determine when such a function initializes class state, even if the initializing function calls other functions to perform parts of the initialization. When the body of the initializer function is not available to PC-lint Plus, such a determination cannot be made. In such cases, you may designate the member as an initializer using the *-sem* option. (The **initializer** semantic is a flag semantic). If a member is designated as an initializer function and the body is available to PC-lint Plus, a complaint will be issued if it fails to initialize all of the data members.

#### cleanup

The **cleanup** semantic does for destructors what **initializer** does for constructors. A function designated as **cleanup** is expected to process each (non-static) member pointer by either freeing it (in any of the various ways of releasing storage) or, at least, zeroing it. Failure to do this will merit Warning 1578. A function that is a candidate for this semantic will be pointed out by Warning 1579. **cleanup** is a flag semantic.

#### inout(*i*)

A semantic expression of the form **inout(*i*)** where *i* is a constant designating a parameter, indicates that an indirect object passed to that parameter will be both read and written by the function. Thus the *i*th parameter must be either a pointer (or, equivalently an array) or a reference.



This should not be used with pointers or references to `const` objects, since, in this case, it is assumed that the object referenced is only read by the function. It is considered an `in` parameter. If the parameter is a pointer or reference to a non-`const` it is assumed by default to be an out parameter. That is, the function will only write to the referenced object but will not read from it.

But there is no linguistic way to deduce that the argument will be both read and written such as, for example, the first argument to `strcat()`. Hence the need for this semantic.

For example:

```
//lint -sem( addto, inout(1) )

void addto( int *p, int b );    // add b to the object pointed to
                                // by the first argument.

void f() {
    int n;
    addto( &n, 12 );          // Warning, n is likely uninitialized
}
```

`custodial(i)` where *i* is some integer denoting the *i*th argument or the letter 't' denoting the `this` pointer.

It indicates that a called function will take 'custody' of a pointer passed to argument *i*. More accurately, it removes the burden of custody from its caller. For example,

```
//lint -sem(push,custodial(1))
void f() {
    int *p = new int;
    push(p);
}
```

Function `f` would normally draw a complaint (Warning 429) that custodial pointer `p` had not been freed or returned. However, with the custodial semantic applied to the first argument of `push`, the call to `push` removes from `f` the responsibility of disposing of the storage allocated to `p`.

To identify the implicit argument of a (non-static) member function you may use the 't' subscript. Thus:

```
//lint -sem( A::push, custodial(t) )
struct A { void push(); ... };
void g( ) {
    A *p = new A;
    p->push();
}
```

You can combine the custodial semantic with a test for NULL. For example,

```
-sem( push, 1p, custodial(1) )
```

will complain about NULL pointers being passed as first argument to `push` as well as giving the custodial property to this argument.

The custodial semantic is an argument semantic meaning that it can be passed on to another function using the argument number as subscript. Thus:

```
function( push(1), append(1) )
```

transfers the custodial property of the 1st argument of `push` (as well as the test for NULL) on to the 1st argument of function `append`. But note you may not transfer `this` semantics using a 0 subscript as that refers to function wide semantics.

An example of the use of the letter `t` to report this is as follows

```
//lint -sem( A::push, custodial(t) )
struct A { void push(); ... };
void g( ) {
    A *p = new A;
    p->push();
}
```

Note that for the purposes of these examples, we have placed the `-sem` options within lint comments. They may also be placed in a project-wide options file (`.lint` file).

`non_custodial(i)` where  $i$  is some integer denoting the  $i$ th argument.

This argument semantic is the opposite of the `custodial` semantic and can be used to indicate that a function does not take custody of the memory pointed to by its argument. When the `ffc` flag is ON (which it is by default), non-library functions that accept pointers to non-const are assumed to take custody of the pointed-to memory. This semantic can be used to individually specify functions for which this assumption is not appropriate. For example:

```
//lint -sem(f, non_custodial(1))
void f(int*);
void g() {
    int* p = new int;
    f(p);
}
```

will report:

```
warning 429: custodial pointer 'p' likely not freed nor returned
```

If the `non_custodial` semantic had not been used, the message would not have been issued because `f` would have taken custody due to the `ffc` flag.

`pod(i)` A semantic expression of the form `pod(i)` where  $i$  is a constant designating a parameter, indicates that the argument is expected to be a pointer to a POD.

A POD is an abbreviation for Plain Old Datatype. In brief, an object of POD can be treated as so many bytes, copyable by `memcpy`, clearable by `memset`, etc. For example:

```
//lint -sem( clear, 1p, pod(1) ) wants a non-null pointer to POD
class A
{ A(); int data; } a;
class B
{ public: int data; } b;
void clear( void *, size_t );
void f() {
    clear( &a, sizeof(a) );    // Warning
    clear( &b, sizeof(b) );    // no Warning
}
```

`pure` This semantic will designate a function as being pure (see definition below).

Normally functions are determined to be pure or impure automatically through an analysis of their definition. However, if a function is external to the source files being linted, this analysis cannot be made and the function is by default considered impure. This semantic can be used to reverse this assumption so that the function is regarded as pure.

The significance of a pure function is that it lacks internal side-effects and this can be used to diagnose code redundancies. There are a number of places in the language (left hand side of a comma, first or third expression of a **for** clause, the expression statement) when it makes no sense to have an expression unless some side-effect is to be achieved. As an example

```
void f() {}
void g()
{
    f();    // Warning 522
}
```

Because we can deduce **f** to be pure, a warning is issued. In general, we may not be aware until pass 1 is finished that a function is pure. You can use the pure semantic to hasten the process of detection.

Another use of this semantic can be to determine on what grounds PC-lint Plus considers a function to be pure. If a function is designated as being pure and is later deemed to have impure properties Warning 453 will be issued with a detailed explanation as to why the function is impure.

Definition of a pure function: A function is said to be pure if it is not impure. A function is said to be impure if it modifies a static or global variable or accesses a volatile variable or contains any I/O operation, or makes a call to any impure function.

A function call is said to have side-effects if it is a call to an impure function or if it is a call to a pure function that modifies its arguments.

Example:

```
int n;
void e1() { n++; }
void e2() { static k; k++ }
void e3() { printf ( "hello" ); }
double e4( double x )
    { return sqrt(x); }
void e5( volatile int k ) { k++; }
void e6() {e1(); }
```

Each of the functions **e1** through **e6** is impure because it satisfies one of the above conditions of being an impure function. (This assumes that both **printf** and **sqrt** are external functions.) On the other hand, in the following:

```
int f1() { int n = 0; n++; return n; }
void f2( int*p ) { *p = f1(); }
```

both **f1** and **f2** are pure functions because there is nothing to designate them impure.

Consider:

```
//lint -sem( sqrt, pure )
void compute()
{
    double x = sqrt( 2.0 );
}
void m()
{ compute(); }
```

Here, because of the **pure** semantic given to **sqrt**, we get a deserved diagnostic (522, Highest operation, function 'compute', lacks side-effects) at the call to **compute**. I'm sure the reader will agree that the function **compute** shows evidence of a lack of completeness. The author may have been side-tracked during development and never got back to completing the function. But as we indicated earlier **sqrt** would by default be considered impure since it is external. It may actually be impure since on error

conditions it needs to set the external variable `errno` to `EDOM`.

Nonetheless, from the standpoint of desired functionability, `compute` comes up short. This can be traced to `sqrt` not offering any desired functionality as a side-effect. Since this is the case, the programmer was justified in inserting the semantic for `sqrt`.

Consider the following example:

```
int f()
{
    int n = 0;
    n++;
    return n;
}
```

`f()` is considered to be a pure function. True it modifies `n` but `n` is an automatic variable. The increment operator is not considered impure but it is regarded as having side-effects.

Consider the following pair of functions:

```
void h(int *p) { (*p)++; }
int g() { int n=0; h(&n); return n;}
```

Here the function `h()` is considered pure but note that the call `h(&n)` has side-effects. Function `g()` is exactly analogous to `f()` above and so must be considered pure. Function `g()` calls upon `h()` to modify variable `n` in much the same way that `f()` earlier employed the increment operator. If `g()` had provided the address of a global variable to `h()` then `g()` would have been considered impure but not `h()`. Had we considered `h()` to be impure irregardless of the nature of its argument then, since `g()` is pure, we would have had to give up the principle that impurity is inherited up the call chain.

**chneg(*i*)** A semantic expression indicating that the *i*th argument is expected to be non-negative.

Calling the function with a negative or possibly negative value will be diagnosed with message [422](#) or [671](#), as appropriate.

**dangerous**

A function designated with the dangerous semantic will cause message [421](#) to be issued when the function is called. This is similar to function deprecation [17.8 Deprecation of Entities](#) but using a semantic allows specific function overloads to be specified.

**noliteral(*i*)**

A parameter with the `noliteral` semantic will report message [2460](#) if the argument is a literal. Message [2960](#) will be reported if the argument is an integer constant expression.

**printf(*i*)**

A semantic expression indicating that this is a `printf`-like function whose format argument is the *i*th argument. An option of the form `-sem(func, printf(i))` is functionally equivalent to the option `-printf(i,func)`.

**scanf(*i*)**

A semantic expression indicating that this is a `scanf`-like function whose format argument is the *i*th argument. An option of the form `-sem(func, scanf(i))` is functionally equivalent to the option `-scanf(i,func)`.

**signal\_register(*i*)**

A semantic expression indicating that this is a [signal handler](#) registration function. The *i*th parameter type should be a function pointer (or a pointer to a structure containing a function pointer) whose argument will be registered as a signal handler.

#### `signal_handler`

Designates a function as a signal handler which is subject to the signal messages [2670](#), [2761](#), [2762](#), [2763](#), and [2765](#). If a function has not been designated with this semantic it may still be implicitly treated as a signal handler due to registration with the `signal` function, the `sigaction` function, or another function with the `signal_register` semantic. If a function is known to be a signal handler within the module in which it is defined, the signal messages will be issued at module wrap-up. If a function is only known to be a signal handler due to its use as an argument to a signal registration function within a module other than the one in which it is defined, the signal messages will be issued at global wrap-up.

#### `exception_signal_handler`

In addition to all behavior specified for the `signal_handler` semantic, the exception signal messages [2671](#) and [2764](#) will be issued for designated exception signal handlers. A signal handler may implicitly be considered an exception signal handler if it was registered to handle SIGBUS, SIGSEGV, SIGFPE, or SIGILL.

#### `async_signal_safe`

An async-signal-safe function can be safely called within a signal handler. Messages [2670](#) and [2761](#) will not be issued for calls to such functions within signal handlers. Unless otherwise specified, a function is neither async-signal-safe nor async-signal-unsafe and message [2761](#) will be issued for calls within signal handlers.

#### `async_signal_unsafe`

An async-signal-unsafe function cannot be safely called within a signal handler. Message [2670](#) will be issued for calls to such functions within signal handlers. Unless otherwise specified, a function is neither async-signal-safe nor async-signal-unsafe and message [2761](#) will be issued for calls within signal handlers.

#### `stream_read(i)`

Specifies that the function reads from the file stream provided as the *i*th argument. Message [2476](#) will be issued if the stream wasn't open for reading. Message [2478](#) will be issued if the previous operation on the stream was by a function with the `stream_write` semantic.

#### `stream_write(i)`

Specifies that the function writes to the file stream provided as the *i*th argument. Message [2477](#) will be issued if the stream wasn't open for writing. Message [2479](#) will be issued if the previous operation on the stream was by a function with the `stream_read` semantic.

#### `stream_push(i)`

Specifies that the function pushes a character back onto the file stream provided as the *i*th argument. Message [2470](#) will be issued for multiple consecutive calls to a function with this semantic using the same stream argument.

#### `stream_pos(i)`

Specifies that the function repositions the file stream provided as the *i*th argument. A call to a function that repositions or flushes a stream is required between read and writes to the same stream.

#### `stream_flush(i)`

Specifies that the function flushes the file stream provided as the *i*th argument. A call to a function that repositions or flushes a stream is required between a write operation and subsequent read.

**stream\_close(*i*)**

Specifies that the function closes the file stream provided as the *i*th argument. Message 2471 is issued for any stream operations performed on a file stream that has been closed.

**stream\_wide(*i*)**

Specifies that the function performs a wide-oriented stream operation on the file stream provided as the *i*th argument. Message 2481 is issued for an attempt to perform a wide-oriented operation on a byte-oriented file stream.

**stream\_byte(*i*)**

Specifies that the function performs a byte-oriented stream operation on the file stream provided as the *i*th argument. Message 2480 is issued for an attempt to perform a byte-oriented operation on a wide-oriented file stream.

**stream\_placid(*i*)**

Specifies that the function which accepts a file stream as the *i*th argument does not perform any operations that modify the state of the stream (state-modifying operations include reading, writing, pushing, repositioning, flushing, and closing). When a file stream is passed to a function taking a pointer to a non-const FILE object, it is assumed that the function modifies the state of the stream if the body of the function is not visible to PC-lint Plus unless the function has the **stream\_placid** semantic.

**no\_ptr\_to\_auto(*i*)**

Specifies that the pointer provided as the *i*th argument should not be a pointer to automatic storage. Violations are reported by message 2601.

**no\_specific\_walk**

Specifies that Value Tracking should skip specific walks of the function. The general walk of the function will still occur. The semantic name **nowalk** is recognized as a historical alias for this semantic for backwards compatibility.

*exp* a semantic expression involving the expression elements described below:

- *in* denotes the *i*th argument, which must be integral (E.g. 3*n* refers to the 3rd argument). An argument is integral if it is typed **int** or some variation of integral such as **char**, **unsigned long**, an enumeration, etc.
- *i* may be @ (commercial at) in which case the return value is implied. For example, the expression:

**@n == 4 || @n > 1n**

states that the return value will either be equal to 4 or will be greater than the first argument.

- *ip* denotes the *i*th argument, which must be some form of pointer (or array). The value of this variable is the number of items pointed to by the pointer (or in the array). For example, the expression:

**2p == 10**

specifies a constraint that the 2nd argument, which happens to be a pointer, should have exactly 10 items. The number of items "pointed to" by a string constant is 1 plus the number of characters between quotes.

Just as with *in*, *i* may be @ in which case the return value is indicated.

- *iP* is like *ip* except that all values are specified in bytes. For example, the semantic:

**2P == 10**

specifies that the size in bytes of the area pointed to by the 2nd argument is 10. To specify a return pointer where the area pointed to is measured in bytes we use `@P`.

- *integer* (any C/C++ integral or character constant) denotes itself.
- *identifier* that refers to a macro that evaluates to a constant expression. The identifier is retained at option processing time and evaluated at the time of function call.
- `malloc( exp )` attaches a `malloc` allocation flag to the expression. See the discussion of Return Semantics below.
- `new( exp )` attaches a `new` allocation flag to the expression.
- `new[] ( exp )` attaches a `new[]` allocation flag to the expression.
- `( )`
- Unary operators: `+` `-` `!` `~`
- Binary operators:  
`+` `-` `*` `/` `%` `<` `<=` `==` `!=` `>` `>=` `|` `&` `^` `<<` `>>` `||` `&&`
- Ternary operator: `?:`

### 9.2.2 Semantic Expressions

Operators, parentheses and constants have their usual C/C++ meaning. Also the precedence of operators are identical to C/C++.

There may be at most two expressions in any one *-sem* option, one expressing Return semantics and one expressing Function-wide semantics.

#### 9.2.2.1 Return Semantics

An expression involving the return value (one of `@n`, `@p`, `@P`) is a Return semantic and indicates something about the returned value. For example, if the semantics for `strlen()` were given explicitly, they might be given as:

```
-sem( strlen, @n < 1p, 1p )
```

In words, the return value is strictly less than the size of the buffer given as first argument. Also the first argument should not be null.

To express further uncertainty about the return value, one or more expressions involving the return value may be alternated using the `||` operator. For example:

```
-sem( fgets, @p == 1p || @p == 0 )
```

represents a possible Return semantic for the built-in function `fgets`. Recall that the `fgets` function returns the address of the buffer passed as first argument unless an end of file (or error) occurs in which case the null pointer is returned. If the Return semantic indicates, in the case of a pointer, that the return value may possibly be zero (by explicitly using either the test `@p == 0` or `@P == 0` as in this example) this is taken as a possibility of returning the null pointer.

As another example:

```
-sem( lookup, 2n == LOCATE ? (@p==0||@p==1) : @p==1)
```

This is a Return semantic that says that if the 2nd argument of the function `lookup` is equal to `LOCATE` then the return pointer may or may not be null. Otherwise we may assume that the return value is a valid non-null pointer. This could be used as follows:

```

#define LOCATE 1
#define INSTALL 2
Symbol *lookup (const char *, int );
...
    sym = lookup("main", INSTALL);
    sym->value=0; /*OK*/
    sym = lookup("help", LOCATE);
    v = sym -> value; /* warning - could be NULL */

```

Here the first return value from lookup is guaranteed to be non-null, whereas the second may be null or may not be.

We caution the reader that the following, apparently equivalent, semantic does not work.

```
-sem( lookup, @p == (2n != LOCATE) || @p == 1 )
```

The OR (||) is taken to mean that either side or both could be true with some probability but there is no certainty deduced that one or the other must be true.

When @p (lowercase p) is used, the pointee type of the return value must be a complete type at the point the function is called since PC-lint Plus needs to be able to calculate the size of the returned buffer to use such a semantic. If the type return type is not complete, a 686 message will be issued.

#### *Flagging the return value*

Consider the example:

```

char *p, *q = 0;
p = malloc(10);
p = q;           // Warning -- memory leak

```

We are able to issue a Warning because the return from malloc has an allocation flag that indicated that the returned value points to a freshly allocated region that is not going to be freed by itself.

The seemingly equivalent semantic option:

```
-sem( my_alloc, @P == 1n )
```

associates no allocation flag with the returned pointer (only the size of the area in bytes).

To identify the kind of storage that a function may return, three flag-endowing functions have been added to the allowed expression syntax of the *-sem* option:

- a region allocated by malloc to be released through free
- a region allocated by new to be released through delete
- a region allocated by new[] to be released through delete[]

In each case, the *exp* is the size of the area to be allocated. For example, to simulate malloc we may have:

```
-sem( my_alloc, @P == malloc(1n) )
```

By contrast the semantic:

```
-sem( some_alloc, @p == malloc(1n) )
```

indicates, because of the lower case 'p', that the size of the allocated region is measured in allocation units. Thus the malloc here is taken to indicate the type of storage (freshly allocated that should be free) and not as a literal call to malloc to allocate so many bytes.

As another example:



```
-sem( newstr, @p == (1p ? new[] (1p) : 0) )
```

In words, this says that `newstr` is a function whose return value will, if the first argument is a non-null pointer, be the equivalent of a `new[]` of that size. Otherwise a NULL will be returned.

### 9.2.2.2 Return Semantic Validation

Return semantics are typically employed on functions for which the body is not available to PC-lint Plus but they can also be applied to functions that do have a visible definition. In this case, the return semantics will be validated against the actual function definition when analyzing specific calls. This can be used, for example, to document the return conditions that the function should employ and to have PC-lint Plus diagnose deviations from this contract.

Violation of return semantics are reported via Warning 2426. In the following example, the semantic `@p > 0` specifies that the pointer return value is never null. The implementation of this function contains a path that violates this semantic. PC-lint Plus will now report when such a path is taken causing the return value semantic to be violated:

```
//lint -sem(f, @p > 0) return value should never be null
void *f(int a, void *p) {
    if (a < 0)
        return 0;
    return p;
}
void g(void *p) {
    void *ptr = f(-1, p);
}
```

PC-lint Plus produces:

```
warning 2426: return value (nullptr) of call to function
             'f(int, void *)' conflicts with return semantic '(@p>0)'
             void *ptr = f(-1, p);
                     ^
```

to indicate the violation of the semantic that specified the return value is never null.

When the return semantic conflicts with information collected during a specific call, the latter overrides the former. For example, despite the existence of the semantic claiming that the return value is not null, after the call to `f` PC-lint Plus will retain the knowledge gleaned from the actual call to `f` and diagnose an attempt to dereference the pointer. For example, if after the call to `f(-1, p)` we had:

```
int *iptr = ptr;
*ptr = 1;
void
```

PC-lint Plus would issue:

```
warning 413: likely use of null pointer
             *iptr = 1;
             ^
supplemental 831: initialization yields nullptr
             int *iptr = ptr;
             ~~~~~^~~~~~
supplemental 831: initialization yields nullptr
             void *ptr = f(-1, p);
             ~~~~~^~~~~~
```

```

supplemental 831: null to pointer conversion yields nullptr
    return 0;
    ~

```

If the `fso` flag is turned ON, return semantics will override any conflicting information obtaining during a specific walk although message 2426 will still be issued.

### 9.2.2.3 Function-wide semantics

An expression that is not a Return semantic is a 'Function-wide' semantic (to use the terminology of Section 9.1.1 [Special Functions](#)). It indicates a predicate that should be true. If there is a decided possibility that it is false, a diagnostic is issued.

What constitutes a "decided possibility"? This is determined by considerations described in Section 8 [Value Tracking](#). If nothing is known about a situation, no diagnostic is issued. If what we do know suggests the possibility of a violation of the Function-wide semantic, a diagnostic is issued.

For example, to check to see if the region of storage passed to function `g()` is at least 6 bytes you may use the following:

```

//lint -sem( g, 1P >= 6 ) 1st arg. must have at least 6 bytes
void g(short *);
void f() {
    short a[3];           // a[] has 6 bytes
    short *p = a + 1;     // p points to 4 bytes
    g(a);    // OK
    g(p);    // Warning
}

```

Several constraints may be AND'ed using the `&&` operator. For example, to check that `fread( buffer, size, count, stream )` has non-zero second and third arguments and that their product exactly equals the size of the buffer you may use the following option.

```
-sem( fread, 1P==2n*3n && 2n>0 && 3n>0 )
```

Note that we rely on C's operator precedence to properly group operator arguments.

To continue with our example we should add Return Semantics. `fread` returns a value no greater than the third argument (`count`). Also, the first and fourth arguments should be checked for null. A complete semantic option for `fread` becomes:

```
-sem( fread, 1P==2n*3n && 2n>0 && 3n>0, @n<=3n, 1p, 4p )
```

It is possible to employ macros in semantic expressions rather than hard numbers. For example:

```

//lint -sem( X::cpy, 1P <= BUFLen )

char *strcpy(char *dest, const char *src);
#define BUFLen 4

class X {
public:
    char buf[BUFLen];
    void cpy(char *p) { strcpy(buf, p); }
    void slen(char *p);
};

void f(X &x) {

```

```

        x.cpy("abcd"); // Warning
        x.cpy("abc");  // OK
    }

```

Just as is the case with *-function*, *-sem* may be applied to member functions. For example:

```

//lint -sem( X::cpy, 1P <= BUFLen )

const int BUFLen = 4;

class X
{
public:
    char buf[BUFLen];
    void cpy( char * p )
        { strcpy( buf, p ); }
    void slen( char * p );
};

void f( X &x )
{
    x.cpy( "abcd" ); // Warning
    x.cpy( "abc" );  // OK
}

```

In this example, the argument to *X::cpy* must be less than or equal to *BUFLen*. The byte requirements of "abcd" are 5 (including the nul character) and *BUFLen* is defined to be 4. Hence a warning is issued here.

To specify semantics for template members, simply ignore the angle brackets in the name given to *-sem*. The semantics will apply to each template instantiation. For example, in the code below the user wants to assign the custodial semantic to the first argument of the *push\_back* function in every instantiation of template *list*. This will avoid a Warning 429 when the pointer is not deleted in *f()*.

```

//lint -sem( std::list::push_back, custodial(1) )

namespace std
{
    template< class T >
        class list
        {
        public:
            void push_back( const int * );
        };
}

std::list<int*> l;

void f()
{
    int *p = new int;
    l.push_back( p ); // OK, push_back takes custody
}

```

#### 9.2.2.4 Overload-Specific Semantics

A user-defined semantic may be applied to a specific function overload by including the function's parameter list in the semantic specification (where a parameter list of (void) represents a function taking no arguments). For example:

```
//lint -sem(foo(int, int), chneg(1))
void foo(int);
void foo(int, int);

void bar() {
    foo(-8);          // Okay
    foo(-8, 20);      // Warning
}
```

A semantic can be applied to a specific function template instantiation by specifying the substituted template parameter types in the function parameter list:

```
//lint -sem(A1::rocker(int, char *, int), 3n <= 2P)
struct A1 {
    template <typename T2>
    int rocker(T2, char *, int);
};

void g() {
    char buf[10];
    A1 a1;
    a1.rocker(1, buf, 20);    // Warning
    a1.rocker(1.2, buf, 20); // Okay
}
```

A semantic can be applied to all templated versions of a function by referencing the names of the template parameters in the argument list:

```
//lint -sem(A1::rocker(T2, char *, int), 3n <= 2P)
struct A1 {
    template <typename T2>
    int rocker(T2, char *, int);
};

void g() {
    char buf[10];
    A1 a1;
    a1.rocker(1, buf, 20);    // Warning
    a1.rocker(1.2, buf, 20); // Warning
}
```

Every function call has 2 or 3 distinct monikers that can be used in a `-sem` option. Since the correct monikers might not be obvious in some scenarios, the monikers associated for each call will be provided via message 879 when the `fsf` flag is ON. If the `fsf` flag was enabled for the above example, the corresponding 879 messages would look like this:

```
info 879: semantic monikers are 'A1::rocker(int, char *, int)',
        'A1::rocker(T2, char *, int)', and 'A1::rocker'
a1.rocker(1, buf, 20);
~

info 879: semantic monikers are 'A1::rocker(double, char *, int)',
        'A1::rocker(T2, char *, int)', and 'A1::rocker'
a1.rocker(1.2, buf, 20);
~
```

The monikers are provided in order of decreasing specificity. The most specific moniker contains the complete parameter list. The next moniker contains the non-substituted template parameter names, this moniker does

not exist for non-template functions. The most generic moniker is just the name of the function.

An overload set may have multiple semantics associated with it although only one semantic will be applied to a given function call, the semantic with the most specific matching function designator. For example:

```
//lint -sem(slow(T1, T1), 1n != 2n)
//lint -sem(slow(int, double), 1n > 0)
//lint -sem(slow, 1n > 1)

template <typename T1>
void slow(T1, T1);

void slow(int, double);
void slow(int);

void h() {
    slow(1, 0);    // Okay
    slow(0, 0);    // Warning, violates semantic for slow(T1, T1)

    slow(1, 3.0);  // Okay
    slow(0, 3.0);  // Warning, violates semantic for slow(int, double)

    slow(2);       // Okay
    slow(1);       // Warning, violates default semantic for slow
}
```

Note the difference between `foo()` and `foo(void)` in a semantic option. The former specifies that the semantic should apply to the C function named `foo` that does not have a prototype whereas the latter specifies a semantic for a function `foo` declared as taking no arguments (either by being declared as `foo(void)` in C or C++ or as `foo()` in C++).

### 9.2.3 Notes on Semantic Specifications

1. Every function has, potentially, a Return semantic (**r**), a Function-wide semantic (**0**), flag semantics (**f**), and Argument semantics for each of the arguments and the implied this argument (**t**). An expression of the form *ip* when it stands alone and is not part of another expression becomes an Argument semantic for argument *i* (presumably a pointer argument). Thus, for the option

```
-sem( f, 2p, 1p > 0 )
```

`2p` becomes an Argument semantic (the pointer should not be NULL) for argument 2. We can transfer this semantic to, say, the 3rd argument of function `g` by using the option

```
-function( f(2), g(3) )
```

The expression `1p>0` becomes the Function-wide semantic for function `f` and can be transferred via the `0` subscript as in:

```
-function( f(0), g(0) )
```

We could have placed these two together as one large semantic as in:

```
-sem( f, 2p && 1p > 0 )
```

The earlier rendition is preferred because there is a specialized set of warning messages for the argument semantic of passing null pointers to functions.

2. Please note that `r_null` and an expression involving argument `@` are Return semantics. You cannot have both in one option. Thus you cannot have

```
-sem( f, r_null, @p = 1p )
```

It is easy to convert this into an acceptable semantic as follows:

```
-sem( f, @p == 0 || @p == 1p )
```

3. The notations for arguments and return values was not chosen capriciously. A notation such as `@n == 2n` may look strange at first but it was chosen so as not to conflict with user identifiers.
4. Please note that the types of arguments are signed integral values. Thus we may write

```
-sem( strlen, @n < 1p )
```

We are not comparing here integers with pointers. Rather we are comparing the number of items that a pointer points to (an integer) with an integral return value.

For uniformity, the arithmetic of semantics is signed integral arithmetic, usually long precision. This means that greater-than comparisons with numbers higher than the largest signed long will not work.

## 10 Metrics

### 10.1 Introduction

Metrics provide access to valuable information about your program and can be used to enforce coding guidelines or measure code quality. A wide variety of statistics are available for a range of subjects including functions, classes, and files.

The two primary forms of output associated with metrics are reports and violations. A metric report provides the values of all metrics, or a nominated subset, in a standard format at the end of analysis. A metric check can be registered which will trigger violation messages when an entity does not meet the conditions of the check.

#### 10.1.1 Terminology

A metric subject represents the abstract notion of all entities of a certain kind. For example, `file` and `function` are metric subjects. A defined list of metrics are associated with each subject, but values of collected metrics are a property of instances of that subject. Each individual file or function (instance) present in the program will collect values for metrics associated with the `file` or `function` subjects respectively.

The value of each metric is either a number, an entity, or a collection of entities. A numeric metric is classified as either a measure or a count. A defined quantity such as Halstead volume or cyclomatic complexity is a measure while a metric such as “number of forward `goto` statements” representing how many entities matching some criteria exist is a count. A collection is an array of instances of a particular metric subject, for example `file.functions` is a collection of metric data for each function defined within a particular file. Collections can be used with built-in operations including `count`, `sum`, and `filter`. See [10.3 Metric Expressions](#) for more details on these operations.

See [10.6 Built-in Metrics](#) for a list of metric subjects and their metrics.

#### 10.1.2 Options

The `+metric_report` option can be used to generate customized reports. The `+metric` option can check for violations of metric-based conditions, nominate metrics for inclusion in the report, and create new custom metrics. All options relating to metrics must be appear prior to the first module.

## 10.2 Metric Report

The `+metric_report` option is used to request a metric report at the end of analysis. By default, all metrics are included in the report. If the `+metric` option has been used to nominate any metrics for explicit inclusion in the report then only nominated metrics will appear. This can be overridden by providing as an argument to the optional `scope` sub-option to `+metric_report` either `all` or `nominated` which determines conclusively whether all metrics are included or only nominated metrics (even if no metrics have been nominated).

The `+metric_report` option accepts the following optional sub-options:

- `scope` — Valid values are `all` and `nominated`. Explicitly determines whether all metrics are included in the report or only nominated metrics. The default behavior is `nominated` if any metrics have been nominated and `all` otherwise.
- `format` — Valid values are `xml`, `json`, or `csv`. The default is `csv`. Note that CSV fields containing literal commas, double quotes, or newlines will be enclosed in double quotes and within such a field literal double quotes will be doubled.
- `filename` — Specifies an output filename where the metric report will be written. If no filename is specified the report will be written to standard error.

For example, the option `+metric_report(scope=all, format=xml, filename=path/file.xml)` will request a report containing all metrics to be written to the file `path/file.xml` in XML format.

### 10.2.1 Metric Nomination

Metrics are nominated using a `+metric` option that simply names a metric using its subject and metric name. For example, `+metric(file.num_lines)` nominates the values of `file.num_lines` to appear on the report for all files.

The metric report includes function and class definitions as well as templates and specializations thereof, but it does not include instantiations.

### 10.2.2 Report Fields

The metric report provides the `subject`, `entity_name`, `id` (metric name), `value`, and `location` for each instance of an included metric. In the CSV format, fields appear in the listed order and column names are not included. In the JSON and XML formats, the indicated names are used and the entries appear enclosed in an outer `metrics` object. The `location` field uses the format `file:line:col` and is filled only for entries with the `function` subject.

## 10.3 Metric Expressions

Metrics use a C-like expression grammar with function calls (to built-in functions), simple assignment, parenthetical grouping, identifiers, numeric literals with optional decimal places, the dot operator, and arithmetic, logical, and comparison operators. Built-in functions are listed below. The assignment operator is available only in the context of defining a custom metric. Identifiers denote the name of a metric or a metric subject, and they may contain letters, underscores, and non-leading digits. Numeric literals are always decimal and consist of a whole number part optionally followed by a period and fractional part. The dot operator takes a metric subject or instance thereof on the left and a metric name on the right; this forms a metric designator which refers to that metric name of that subject, analogous to structure access in C. An expression is considered to “evaluate to true” if the result is a non-zero numeric value. Evaluation errors may yield an empty value and evaluation of a sub-expression to an empty value will typically cause evaluation of the enclosing expression to fail in turn. When a metric subject appears in an expression, it generally leads to implicit iteration in the enclosing context where the expression will be executed multiple times, once with each instance of that subject substituted in place of the abstract subject.

### 10.3.1 Built-in Functions

- `count(collection)` (numeric) — Returns the size of the provided collection.
- `sum(collection, expression)` (numeric) — Returns the sum of the results of evaluating the provided expression for each item in the provided collection.
- `average(collection, expression)` (numeric) — Returns the arithmetic mean of the results of evaluating the provided expression for each item in the provided collection.
- `median(collection, expression)` (numeric) — Returns the median of the results of evaluating the provided expression for each item in the provided collection.
- `min(collection, expression)` (numeric) — Returns the minimum value of the provided expression for any item in the provided collection.
- `max(collection, expression)` (numeric) — Returns the maximum value of the provided expression for any item in the provided collection.
- `min_item(collection, expression)` (entity) — Returns the entity in the provided collection for which the minimum value of the provided expression occurred.



- `max_item(collection, expression)` (entity) — Returns the entity in the provided collection for which the maximum value of the provided expression occurred.
- `filter(collection, condition)` (collection) — Returns a new collection containing the items from the provided collection for which the provided condition evaluates to true.
- `ln(number)` (numeric) — Evaluates the natural logarithm,  $\log_e x$ , of the provided expression.
- `exp(number)` (numeric) — Evaluates the natural exponential function,  $e^x$ , of the provided expression.
- `abs(number)` (numeric) — Evaluates the absolute value,  $|x|$ , of the provided expression.

## 10.4 Metric Checking

If the expression provided to a `+metric` option is a comparison operator (possibly within a logical operator) then it introduces a new metric check. The expression must contain at least one metric designator and the subject (left component) of all metric designators in the top-level expression must match. Multiple metric names for the same subject may appear. The provided expression will be evaluated at the end of analysis for each instance of that subject (e.g. for the subject `file` the expression would be evaluated separately using the metric values for each file). The check is considered to be violated if the expression evaluates to true. A message in the `info` category will be issued reporting this violation. By default this will be `888`, although this is configurable using the `msgno` sub-option.

As an example, the option `+metric(function.num_return_stmts > 1)` will cause a message to be emitted when a function contains multiple `return` statements. If a function `f` with two `return` statements is defined in the presence of this option then PC-lint Plus will emit:

```
info 888: number of return statements (2) in function 'f' is greater than 1
```

Multiple metric designators involving the same subject may appear, for example `+metric(file.num_comment_lines / file.num_lines < 0.10)` will cause a message to be emitted if fewer than 10% of lines in a file are comment lines (see Definitions below for the precise specification of terms like “comment line”).

When registering a metric check, the `+metric` option accepts the following sub-options:

- `msgno` — Specifies a custom message number to use when reporting a violation instead of `888`. The criteria for custom message numbers are the same as for the `+message` option.
- `append` — Specifies text to append to the end of the message when reporting a violation. This text will be appended prior to any text that may also be appended based on the message number by the general `-append` option.

For example, violations reported in response to the option `+metric(file.num_lines >= 100 && file.num_comment_lines < 10, msgno=8042, append=[example])` will be emitted with message number 8042 and `[example]` appended to the message text.

Metric checks are generally performed during or after global wrap-up and can incorporate information from multiple modules. In unit checkout mode, metric checks for functions, classes, and modules are performed locally at module wrap-up. Metric data for each module is independent when performing metric checks in unit checkout mode.

### 10.4.1 Metric Violation Messages

Metric violation messages are generated automatically based on the provided expression and the circumstances of the individual violation. The previous example demonstrates that a violation involving the expression `function.num_return_stmts > 1` can produce a message rendered as `number of return statements (2) in function 'f' is greater than 1` (where 2 is the number of `return` statements present in `f`).

When using a more complex expression, the message text may change not only in value parameterization but also in form. For example:

```
+metric(
  (function.num_backward_goto_stmts > 0) ||
  (function.num_forward_goto_stmts > 0 && (function.num_labels > 1 || !function.isExternC))
  ,msgno=8001
)
```

This check enforces that `goto` statements may only be used for the purposes of jumping to a single cleanup label at the end of an `extern "C"` function. Different violations of this requirement produce different messages. For example:

```
extern "C" void f() {
    goto clean;
clean:
    return;
}
```

does not violate the check.

```
void w() {
begin:
    goto begin;
}
```

violates the check because `w` contains a backwards `goto` statement which is never permitted. This emits:

```
info 8001: number of backward goto statements (1) in function 'w' is greater than 0
```

In the alternative example:

```
void h() {
    goto clean;
clean:
    return;
}
```

the message is again reported but the text is now:

```
info 8001: number of forward goto statements (1) in function 'h' is greater than 0
          and not isExternC of function 'h'
```

which highlights a different portion of the requirement.

#### 10.4.2 Integration with Queries

Metric checks for a class or function can utilize [Queries](#) to access AST information. Single argument [Query Functions](#) that match the metric subject can be accessed by name in the same manner as a built-in metric. For example, the previously discussed option:

```
+metric(
  (function.num_backward_goto_stmts > 0) ||
  (function.num_forward_goto_stmts > 0 && (function.num_labels > 1 || !function.isExternC))
)
```

creates a metric check which will report general usage of `goto` while allowing `extern "C"` functions to follow a permitted pattern of exception handling using a cleanup label. This uses the query function `isExternC` to determine whether the function is declared as `extern "C"`.

## 10.5 Custom Metrics

If the expression provided to a `+metric` option is an assignment then the left operand must be a metric designator and the right operand must be an expression which may include other metric designators referring to different metrics about the same subject as the assignment target.

Custom metrics may be defined in terms of other custom metrics, but evaluation of a custom metric recursively on the same object will cause an error. Note that unevaluated recursion is permitted; the computation of a custom metric may involve evaluation of the same custom metric of a different object if it is defined in a way that leads to a base case. See the [10.5.1.2 Number of inheritance edges in class hierarchy](#) example below for an application of recursion.

Metric designators referring to custom metrics can be nominated or used in checks like built-in metrics. A custom metric cannot redefine a built-in metric nor a previously defined custom metric.

Assignments may not appear (and by extension, custom metrics may not be created) within a larger expression of any `+metric` option. A custom metric can only be created when a `+metric` option specifies exactly one top-level assignment.

The sub-option `nominate`, taking no arguments, may be specified to additionally nominate the newly created metric in a manner equivalent to an explicit `+metric` nomination option referring to the metric designator being assigned to.

### 10.5.1 Examples

#### 10.5.1.1 Percentage of functions in a file with multiple return statements

`return_ratio.lnt:`

```
// Create a new custom metric called 'funcs_with_multiple_returns' for any 'file'.
// This custom metric is a collection and consists of all functions from the built-in
// 'file.functions' where the condition 'function.num_return_stmts > 1' is true.
+metric(file.funcs_with_multiple_returns = filter(file.functions, function.num_return_stmts > 1))
// Create a new custom metric called 'multiple_return_ratio' for any 'file'.
// This custom metric is numeric and is the ratio of the sizes of the
// 'funcs_with_multiple_returns' and 'functions' collections. The size of a collection is found
// using the 'count' function.
+metric(file.multiple_return_ratio = count(file.funcs_with_multiple_returns) / count(file.functions))
// Nominate 'multiple_return_ratio' to appear on the report.
+metric(file.multiple_return_ratio)
// Request a metric report. It will be emitted to standard error at the end of analysis.
// Because a metric was nominated, only nominated metrics will be displayed.
// Only 'multiple_return_ratio' has been nominated. The default format is CSV.
+metric_report
```

`test.cpp:`

```
int f1(int a) { return a + 1; }
int f2(int a) { return a + 2; }
int f3(int a) {
    if (a == 5) {
        return 3;
    } else {
        return 300;
    }
}
int f4(int a) { return a + 4; }
```

**Example report output:**

```
file,test.cpp,multiple_return_ratio,0.250
```

**10.5.1.2 Number of inheritance edges in class hierarchy**

For example, consider the class hierarchy:

```
struct X { };
struct Y : X { };
struct Z { };
struct A : Z, Y { };
struct B { };
struct C : A, B { };
struct D : C { };
```

To compute the total number of inheritance edges in the hierarchy for each class, a custom metric can be created:

```
+metric(
    class.hierarchy_edges =
        count(class.immediate_bases) +
        sum(class.immediate_bases, class.hierarchy_edges)
)
```

This option defines the custom metric `class.hierarchy_edges` recursively to the sum of the number of immediate base classes of all base classes in the hierarchy. The recursion eventually ends when a class has no immediate bases, which yields a `sum` of 0 without evaluating `hierarchy_edges`. This produces the report:

```
class,A,hierarchy_edges,3
class,B,hierarchy_edges,0
class,C,hierarchy_edges,5
class,D,hierarchy_edges,6
class,X,hierarchy_edges,0
class,Y,hierarchy_edges,1
class,Z,hierarchy_edges,0
```

**10.5.1.3 Ratio of number of lines in member function to average number of lines in member functions of the containing class**

This example creates a custom metric, `metric_length_ratio`, representing the ratio of the number of lines in a member function to the average number of lines in member functions of the containing class. The first option creates a custom metric called `avg_method_lines` which calculates the average number of lines across methods in a class. The second option uses this `avg_method_lines` metric from the enclosing class to define `method_length_ratio` for each function.

```
+metric(class.avg_method_lines = average(class.non_static_member_functions, function.num_lines))
+metric(function.method_length_ratio = function.num_lines / function.parent_class.avg_method_lines)
```

**10.6 Built-in Metrics****10.6.1 Definitions**

- **Comment line** – A comment line is a line that contains any portion of a comment. This includes blank lines within C-style comments as well as lines that consist entirely of a `/*` or `//` comment introducer or `*/` comment terminator if they are part of a comment.
- **Code Line** – A code line is a line that contains one or more non-whitespace characters that appear outside of comments.

- **Empty Line** – An empty line is a line outside of a comment that does not contain any non-whitespace characters.
- **Line** – The total number of lines in a file is counted using the POSIX definition of a line, but non-empty files that do not end in a newline are processed as if a trailing newline had been inserted.

Halstead metrics, from Halstead’s “software science”, are defined by equations based on the classification of program tokens as operators or operands [5].

Note that in C++, the standard grammar term *function-try-block* refers to a construct of the form:

```
void f() try { } catch (...) { }
```

where the compound statement typically encompassing a function body is replaced with a top-level **try-catch**.

### 10.6.2 Project (project)

- **defined classes** – **classes** (collection)  
A collection of classes defined in the project. This includes all **class/struct/union** definitions arising from any alternative preprocessing branches or macro definitions that could arise when a file is included multiple times or in multiple translation units.
- **defined functions** – **functions** (collection)  
A collection of functions defined in the project. This includes all function definitions arising from any alternative preprocessing branches or macro definitions that could arise when a file is included multiple times or in multiple translation units.
- **number of distinct files** – **num\_distinct\_files** (count)  
The number of distinct files processed across all translation units. Each source or header file is counted once even if it is utilized multiple times.  
The identity of a single file referenced using different paths will generally be consolidated, but this is not necessarily possible if a single logical file is inconsistently referenced through link, junction, mount point, or network filesystem arrangements.
- **number of elective messages emitted** – **num\_elective\_messages\_emitted** (count)  
The number of messages emitted in the elective note category. See [10.6.2 number of messages emitted](#) for general information.
- **number of error messages emitted** – **num\_error\_messages\_emitted** (count)  
The number of messages emitted in the error category. See [10.6.2 number of messages emitted](#) for general information.
- **number of info messages emitted** – **num\_info\_messages\_emitted** (count)  
The number of messages emitted in the informational category. See [10.6.2 number of messages emitted](#) for general information.
- **number of messages emitted** – **num\_messages\_emitted** (count)  
The number of messages emitted is only available on the metric report. It cannot be used in a metric check. The count includes all messages issued prior to the report. Messages 870 and 900 may appear after the report has already been emitted, and they will not be included in the count.
- **number of physical code lines** – **num\_physical\_code\_lines** (count)  
The number of physical [code lines](#) in the project is the total number of code lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.
- **number of physical comment lines** – **num\_physical\_comment\_lines** (count)

The number of physical [comment lines](#) in the project is the total number of comment lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.

- **number of physical lines** – `num_physical_lines` (count)

The number of physical lines in the project is the total number of lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.

- **number of recursive cycles** – `num_recursive_cycles` (count)

The number of independent recursive cycles in the project call graph.

For example:

```
void f() { f(); }
void g() { g(); }
```

contains two recursive cycles, and:

```
void a(), b(), c();
void a() { b(); }
void b() { c(); }
void c() { a(); }
```

contains one recursive cycle.

Recursive cycles must be independent:

```
void x(), y(), z();
void x() { y(); z(); }
void y() { x(); }
void z() { x(); }
```

contains only one recursive cycle.

This value will exclude recursive cycles in library code unless [fml](#) and [flf](#) are enabled.

- **number of supplemental messages emitted** – `num_supplemental_messages_emitted` (count)

The number of messages emitted in the supplemental category. See [10.6.2 number of messages emitted](#) for general information.

- **number of translation units** – `num_translation_units` (count)

The number of translation units in a project.

- **number of warning messages emitted** – `num_warning_messages_emitted` (count)

The number of messages emitted in the warning category. See [10.6.2 number of messages emitted](#) for general information.

- **translation units** – `translation_units` (collection)

This collection includes all of the translation units (modules) within the project.

### 10.6.3 Translation Unit (`translation_unit`)

- **main file** – `main_file` (entity)  
The file associated with the main file of this translation unit. This is the C or C++ file that was provided as an argument and processed as a module.
- **maximum include depth** – `max_include_depth` (count)  
The maximum depth of nested `#include` directives. A source file with no `#include` directives has a value of zero. A source file that includes only headers that do not themselves contain `#include` directives has a value of one. A source file that includes a header that itself includes a second header has a value of two, and so on.
- **number of distinct files** – `num_distinct_files` (count)  
The number of distinct files in a translation unit. Each source or header file is counted once even if it is utilized multiple times.  
The identity of a single file referenced using different paths will generally be consolidated, but this is not necessarily possible if a single logical file is inconsistently referenced through link, junction, mount point, or network filesystem arrangements.
- **number of physical code lines** – `num_physical_code_lines` (count)  
The number of physical [code lines](#) in a translation unit is the total number of code lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.
- **number of physical comment lines** – `num_physical_comment_lines` (count)  
The number of physical [comment lines](#) in a translation unit is the total number of comment lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.
- **number of physical lines** – `num_physical_lines` (count)  
The number of physical lines in a translation unit is the total number of lines in all distinct files. Note that “distinct” is used with regard to the files whose lines are counted, not the content of the lines.

### 10.6.4 File (`file`)

- **defined classes** – `classes` (collection)  
A collection of classes defined in a file. This includes all class definitions arising from any alternative preprocessing branches or macro definitions that could arise when a file is included multiple times.
- **defined functions** – `functions` (collection)  
A collection of functions defined in a file. This includes all function definitions arising from any alternative preprocessing branches or macro definitions that could arise when a file is included multiple times.
- **Halstead bugs** – `halstead_bugs` (measure)  
The Halstead bugs delivered,  $B$ , of the original text of a single file before preprocessing.
- **Halstead difficulty** – `halstead_difficulty` (measure)  
The Halstead difficulty,  $D$ , of the original text of a single file before preprocessing.
- **Halstead effort** – `halstead_effort` (measure)  
The Halstead effort,  $E$ , of the original text of a single file before preprocessing.
- **Halstead length** – `halstead_length` (measure)

The Halstead length,  $N$ , of the original text of a single file before preprocessing.

- **Halstead level** – `halstead_level` (measure)

The Halstead level,  $L$ , of the original text of a single file before preprocessing.

- **Halstead time** – `halstead_time` (measure)

The Halstead time,  $T$ , of the original text of a single file before preprocessing.

- **Halstead vocabulary** – `halstead_vocabulary` (measure)

The Halstead vocabulary,  $\eta$ , of the original text of a single file before preprocessing.

- **Halstead volume** – `halstead_volume` (measure)

The Halstead volume,  $V$ , of the original text of a single file before preprocessing.

- **number of code lines** – `num_code_lines` (count)

The number of [code lines](#) in a file prior to preprocessing.

- **number of comment lines** – `num_comment_lines` (count)

The number of [comment lines](#) in a file prior to preprocessing.

- **number of empty lines** – `num_empty_lines` (count)

The number of [empty lines](#) in a file prior to preprocessing.

- **number of include directives** – `num_include_directives` (count)

The number of lines with the grammatical form of an include directive written within a file prior to preprocessing. This includes directives within conditionally disabled regions.

- **number of lines** – `num_lines` (count)

The number of [lines](#) in a file prior to preprocessing.

#### 10.6.5 Function (`function`)

- **average scope nesting depth** – `avg_scope_nesting_depth` (measure)

The average nesting depth of compound statements enclosing each statement in a function. That is, the sum for each integral nesting level of the product of that nesting level and the number of statements with that nesting level, divided by the number of statements.

- **cyclomatic complexity** – `cyclomatic_complexity` (measure)

The McCabe cyclomatic complexity of a function.

- **Halstead bugs** – `halstead_bugs` (measure)

The Halstead bugs delivered,  $B$ , of the text of a function.

- **Halstead difficulty** – `halstead_difficulty` (measure)

The Halstead difficulty,  $D$ , of the text of a function.

- **Halstead effort** – `halstead_effort` (measure)

The Halstead effort,  $E$ , of the text of a function.

- **Halstead length** – `halstead_length` (measure)

The Halstead length,  $N$ , of the text of a function.

- **Halstead level** – `halstead_level` (measure)

The Halstead level,  $L$ , of the text of a function.



- **Halstead time** – `halstead_time` (measure)  
The Halstead time,  $T$ , of the text of a function.
- **Halstead vocabulary** – `halstead_vocabulary` (measure)  
The Halstead vocabulary,  $\eta$ , of the text of a function.
- **Halstead volume** – `halstead_volume` (measure)  
The Halstead volume,  $V$ , of the text of a function.
- **maximum cases in switch** – `max_cases_in_switch` (count)  
The maximum number of cases for a single `switch` statement within a function. The `default` label, if present, is included in the number of cases.
- **maximum catch handlers in try** – `max_catch_handlers_in_try` (count)  
The maximum number of catch handlers for a single `try` statement within a function.
- **maximum if else chain paths** – `max_if_else_chain_paths` (count)  
The maximum number of paths through any single `if...else` chain within a function. An `if...else` chain is a continuous structure of the form:

```
if (a) { /* ... */ }
else if (b) { /* ... */ }
else if (c) { /* ... */ }
else { /* ... */ }
```

In this example, there are four paths as control may pass through `a`, `b`, `c`, or `else`. Note that if the final `else` is deleted, the number of paths is still four because control may pass through `a`, `b`, `c`, or none.

An isolated `if`:

```
if (a) { /* ... */ }
```

has two paths.

- **maximum loop nesting depth** – `max_loop_nesting_depth` (count)  
The maximum number of nested `for`, `while`, and `do` statements in a function. A single top-level loop statement yields a depth of 1.
- **maximum scope nesting depth** – `max_scope_nesting_depth` (count)  
The maximum depth of compound statement nesting in a function. The top-level compound statement of an empty function (or the top-level compound statements of a *function-try-block*) have a depth of 1.
- **NPATH** – `npath` (measure)  
The Nejmeh NPATH metric measures the exponential combinations of possible paths through a function. It is distinguished from cyclomatic complexity by its sensitivity to nesting.
- **number of backward goto statements** – `num_backward_goto_stmts` (count)  
The number of backward `goto` statements within a function. A backward `goto` statement is one where the destination label occurs before the `goto` statement.

This includes instances where a `goto` statement returns to an earlier label such as:

```
LABEL1:
    x += 5;
    if (x > 100) { break; }
    goto LABEL1;
```

as well as cases where the `goto` statement targets a label of which it is a sub-statement, e.g.:

```

LABEL2:
    goto LABEL2;

```

Uses of the non-standard “address of label” (`void* p = &&LABEL;`) and “computed goto” (`goto *p;`) language extensions are not counted.

- **number of called functions** – `num_called_functions` (count)  
The number of distinct functions defined within the analyzed project called by this function. Each function called at least once is only counted once regardless of how many different calls to it appear within the body of this function. This metric does not count called functions existing only as declarations that are never defined anywhere within the analyzed project.
- **number of calling functions** – `num_calling_functions` (count)  
The number of distinct functions that contain at least one call to this function. Each function containing at least one call to this function is only counted once regardless of how many different calls to this function appear within the body.
- **number of calls** – `num_calls` (count)  
The number of calls within a function definition. This includes all direct, explicit function calls written within the function body. This includes calls to member functions and calls through function pointers. It does not include invocations of constructors or destructors. Calls to overloaded operators are not included when invoked implicitly using operator syntax. Multiple distinct call sites that each call the same destination function are counted as separate calls.
- **number of catch handlers** – `num_catch_handlers` (count)  
The total number of catch handlers across all `try` statements within a function. This includes `catch(...)` handlers as well as any handlers of a *function-try-block*.
- **number of code lines** – `num_code_lines` (count)  
The number of [code lines](#) in a lexical function definition.
- **number of comment lines** – `num_comment_lines` (count)  
The number of [comment lines](#) in a lexical function definition.
- **number of control statements** – `num_control_stmts` (count)  
The number of a control statements within a function. A control statement refers to any of: `do`, `if`, `switch`, `for`, or `while`.
- **number of do statements** – `num_do_stmts` (count)  
The number of `do` statements within a function.
- **number of uses of else** – `num_else` (count)  
The number of uses of `else` within a function. All uses of `else` are counted (including uses in the `else if` pattern).
- **number of empty lines** – `num_empty_lines` (count)  
The number of [empty lines](#) in a lexical function definition.
- **number of expression statements** – `num_expr_stmts` (count)  
The number of expression statements within a function. This is a subset of [Number of Statements](#) that includes only those statements which are expressions (as opposed to declarations or control statements).
- **number of for statements** – `num_for_stmts` (count)  
The number of `for` statements within a function. All `for` statements are counted, including ranged-based `for`.

- number of forward goto statements** – `num_forward_goto_stmts` (count)  
 The number of forward `goto` statements within a function. A forward `goto` statement is one where the destination label occurs after the `goto` statement.  
 Uses of the non-standard “address of label” (`void* p = &&LABEL;`) and “computed goto” (`goto *p;`) language extensions are not counted.
- number of global variables referenced** – `num_global_vars_referenced` (count)  
 The number of distinct global variables referenced within a function. This counts the number of distinct variables that are referenced at least once without regard for the number of times a single distinct global variable may be repeatedly referenced.
- number of if statements** – `num_if_stmts` (count)  
 The number of `if` statements within a function. All `if` statements are counted (including those used in the `else if` pattern).
- number of incoming calls** – `num_incoming_calls` (count)  
 The number of distinct, direct calls to this function. This is the number of call sites where this function is called directly (not through an intermediate function pointer). Multiple written calls to this function in the body of a single function are each counted as separate call sites. Recursive call sites (calls made within this function itself) are included.
- number of instance data members read** – `num_instance_data_members_read` (count)  
 For a non-static member function, the number of distinct non-static instance data members read in the function body.
- number of instance data members referenced** – `num_instance_data_members_referenced` (count)  
 For a non-static member function, the number of distinct non-static instance data members referenced in the function body.
- number of instance data members written** – `num_instance_data_members_written` (count)  
 For a non-static member function, the number of distinct non-static instance data members written to in the function body.
- number of jump statements** – `num_jump_stmts` (count)  
 The number of `return`, `break`, `continue`, and `goto` statements within a function.
- number of labels** – `num_labels` (count)  
 The number of identifier labels within a function. This does not include `case` or `default` labels.
- number of lines** – `num_lines` (count)  
 The number of [lines](#) in a lexical function definition.
- number of loops** – `num_loops` (count)  
 The number of `while`, `for`, and `do` statements within a function.
- number of monocarpic do statements** – `num_monocarpic_do_stmts` (count)  
 The number of `do...while` statements that serve only to group statements because the controlling expression is `false` or `0`. See message [717](#).
- number of mutable instance data members read** – `num_mutable_instance_data_members_read` (count)  
 For a non-static member function, the number of distinct non-static mutable data members read in the function body.

- **number of mutable instance data members written** – `num_mutable_instance_data_members_written` (count)  
For a non-static member function, the number of distinct non-static mutable data members written to in the function body.
- **number of non-static calls with base other than this** – `num_non_static_calls_non_this` (count)  
The number of calls to non-static member functions on objects other than ‘this’. This does not include constructors, destructors, or invocations through an overloaded operator.
- **number of non-static local variables** – `num_non_static_local_vars` (count)  
The number of non-static local variables defined within a function. This does not include parameters.
- **number of loops with facially non-terminating conditions** – `num_non_term_cond_loops` (count)  
The number of `for`, `while`, and `do` statements within a function that have constant conditions of `1` or `true`. This also includes `for` statements with the condition omitted.  
These loops may or may not contain control flow to escape from an otherwise infinite loop.
- **number of parameters** – `num_parameters` (count)  
The number of parameters for a function. An ellipsis denoting variable arguments does not affect the value. The implicit object argument (`this`) of a member function is not counted. Parameters with default arguments are counted.
- **number of parameters with default arguments** – `num_parameters_with_default_args` (count)  
The number of parameters with default arguments for a function.
- **number of return statements** – `num_return_stmts` (count)  
The number of `return` statements within a function. The implicit return at the end of `main` or a function defined as returning `void` (or a `catch` clause of a *function-try-block* thereof) is not included. A `return` statement within a lambda body or a method of a local class is not attributed to the enclosing function.
- **number of selection statements** – `num_selection_stmts` (count)  
The number of `if` and `switch` statements within a function.
- **number of static local variables** – `num_static_local_vars` (count)  
The number of static local variables defined within a function. This includes `thread_local` variables.
- **number of statements** – `num_stmts` (count)  
The number of statements within a function. Every statement within the body of any compound statement is counted. Two important implications of this methodology are:
  1. Grammatically, a label (named, `case`, or `default`) is a “labeled statement” and the statement that it labels is a substatement. A labeled statement is only counted as a single statement, regardless of label nesting depth. E.g. `case 1: case 2: case 3: break;` is counted as one statement, not four.
  2. A selection or iteration statement whose body is not a compound statement is counted as a single statement. E.g. `if (x) return;` is counted as one statement, not two. If the body is a compound statement, then each statement therein will be counted as well as the control statement itself.
- **number of switch cases** – `num_switch_cases` (count)  
The total number of `case` and `default` cases across all `switch` statements within a function.
- **number of switch statements** – `num_switch_stmts` (count)

The number of `switch` statements within a function.

- **number of switch statements without default** – `num_switch_stmts_without_default` (count)  
The number of `switch` statements without a `default` within a function. All `switch` statements without a written `default` are counted. A `switch` statement that covers its entire integral or enumeration type domain with `case` labels but has no written `default` will still be counted.
- **number of throw exprs** – `num_throw_exprs` (count)  
The number of `throw` expressions within a function.
- **number of try statements** – `num_try_stmts` (count)  
The number of `try` statements within a function. This includes a *function-try-block*.
- **number of while statements** – `num_while_stmts` (count)  
The number of `while` statements within a function. This does not include the appearance of the keyword `while` in the syntax of a `do` statement condition.
- **parent class** – `parent_class` (entity)  
For a non-static member function, `parent_class` refers to the class of which it is a member.

#### 10.6.6 Class (class)

- **immediate bases** – `immediate_bases` (collection)  
This collection includes all of the immediate base classes of a class. Immediate base classes are those that the class inherits from directly. If `X` inherits from `Y` and `Y` inherits from `Z`, then `Y` is an immediate base of `X` and `Z` is an immediate base of `Y`, but `Z` is not an immediate base of `X`.  
To see how this can be used recursively to define custom metrics that consider an entire base class hierarchy, refer to [10.7.1 Levels of Inheritance to Most Distant Base](#).
- **immediate children** – `immediate_children` (collection)  
This collection includes all of the immediate derived (child) classes of a class. Immediate derived classes are those that inherit from the class directly. If `X` inherits from `Y` and `Y` inherits from `Z`, then `X` is an immediate child of `Y` and `Y` is an immediate child of `Z`, but `X` is not an immediate child of `Z`.  
To see how this can be used recursively to define custom metrics that consider an entire derived class hierarchy, refer to [10.7.1 Depth of Inheritance Tree](#).
- **instance data implementation partition** – `instance_data_implementation_partition` (measure)  
The instance data implementation partition measure conveys the independence or entanglement of groups of non-static data members. It is scaled from 0 to 100 and is calculated based on the relative difference of the number of independent groups of inseparable non-static data members and the total number of non-static data members. All non-static data members referenced within any single member function are inseparable, and inseparability is transitive. For example, in:

```
class X {
public:
    int f1() { return a + b; }
    void f2() { b = c; }
    void f3() { d = 0; }
    int f4() { return e ? f : g; }
private:
    int a;
    int b;
    int c;
    int d;
```

```

    int e;
    int f;
    int g;
};

```

The independent, inseparable groups are:

- a, b, c
- d
- e, f, g

With seven non-static data members ( $M$ ) and three inseparable groups ( $G$ ), the measure evaluates to  $100 \frac{M-G}{M-1} = 100 \frac{7-3}{7-1} \approx 67$ .

If a class only referenced at most one of its non-static data members per member function then the measure would be 0. If a class had a single member function that referenced all of its non-static data members, the measure would be 100. The measure of a typical class illustrates where it falls between these two extremes.

Note that non-static data members that are not referenced in any of the considered non-static member functions are not included in  $M$  nor  $G$ . References within constructors, destructors, and assignment operators are ignored.

- **lack of cohesion in methods** – `lack_of_cohesion_in_methods` (measure)

Lack of cohesion in methods is the Chidamber-Kemerer LCOM metric. Each combination of two methods is classified based on whether they access any common instance data members, and the value of this metric is the number of such pairs that do not minus the number of such pairs that do. Negative values are clamped to zero.

- **non-static member functions** – `non_static_member_functions` (collection)

A collection of non-static member functions in a class.

- **number of code lines** – `num_code_lines` (count)

The number of [code lines](#) in a lexical class definition.

- **number of comment lines** – `num_comment_lines` (count)

The number of [comment lines](#) in a lexical class definition.

- **number of const member functions** – `num_const_member_functions` (count)

The number of non-static member functions declared with the `const` qualifier. Member functions of base classes are not included.

- **number of empty lines** – `num_empty_lines` (count)

The number of [empty lines](#) in a lexical class definition.

- **number of inline definition member functions** – `num_inline_def_member_functions` (count)

The number of non-static member functions with inline definitions provided within the class definition. This includes member functions defined within the class definition even without an explicit use of the `inline` keyword. Member functions of base classes are not included. User-declared constructors and destructors are included, while special member functions implicitly declared by the compiler are not.

- **number of lines** – `num_lines` (count)

The number of [lines](#) in a lexical class definition.

- **number of mutable data members** – `num_mutable_data_members` (count)

The number of mutable non-static data members. Data members of base classes are not included.

- **number of non-static data members** – `num_non_static_data_members` (count)  
The number of non-static data members within a class. Data members inherited from base classes are not included.
- **number of non-static member functions** – `num_non_static_member_functions` (count)  
The number of non-static member functions in a class. Member functions of base classes are not included. User-declared constructors and destructors are included, while special member functions implicitly declared by the compiler are not.
- **number of pointer data members** – `num_pointer_data_members` (count)  
The number of non-static data members of pointer type. A data member of type reference to pointer is not counted as a pointer. Data members of base classes are not included.
- **number of private data members** – `num_private_data_members` (count)  
The number of non-static data members with private access. Data members of base classes are not included.
- **number of private member functions** – `num_private_member_functions` (count)  
The number of non-static member functions with private access. Member functions of base classes are not included. User-declared constructors and destructors are included, while special member functions implicitly declared by the compiler are not.
- **number of protected data members** – `num_protected_data_members` (count)  
The number of non-static data members with protected access. Data members of base classes are not included.
- **number of protected member functions** – `num_protected_member_functions` (count)  
The number of non-static member functions with protected access. Member functions of base classes are not included. User-declared constructors and destructors are included, while special member functions implicitly declared by the compiler are not.
- **number of public data members** – `num_public_data_members` (count)  
The number of non-static data members with public access. Data members of base classes are not included.
- **number of public member functions** – `num_public_member_functions` (count)  
The number of non-static member functions with public access. Member functions of base classes are not included. User-declared constructors and destructors are included, while special member functions implicitly declared by the compiler are not.
- **number of public virtual member functions** – `num_public_virtual_member_functions` (count)  
The number of virtual non-static member functions with public access. Member functions of base classes are not included. Public virtual destructors are included.
- **number of pure virtual member functions** – `num_pure_virtual_member_functions` (count)  
The number of pure virtual non-static member functions. Pure virtual functions for which an out of class definition is provided are included. Member functions of base classes are not included.
- **number of reference data members** – `num_reference_data_members` (count)  
The number of non-static data members of reference type. Data members of base classes are not included.
- **number of virtual member functions** – `num_virtual_member_functions` (count)  
The number of virtual non-static member functions. Member functions of base classes are not included. Virtual destructors are included.

- **response for class – response\_for\_class** (measure)

Response for class is the Chidamber-Kemerer RFC metric. This is total number of distinct functions in a set containing all methods of the class and all functions directly called by methods of the class.

Constructors, destructors, assignment operators, and implicit special member functions are not considered to be methods. Invocations of overloaded operators or destructors within methods do not contribute to the response set.

### 10.6.7 Dynamic Metrics

Project hosts provide dynamic metrics of the form `num_message_N_emitted` where N is a message number emitted at least once during analysis. For example, if two instances of message 9001 are emitted, then the dynamic metric `project.num_message_9001_emitted` will have a value of 2 on the report. Dynamic metrics are not available in metric checks.

## 10.7 Sample Derived Metrics

The flexibility of metric expressions provides wide latitude to derive ratios, averages, and entirely new metrics from the data listed in in the previous section. The definitions of common derived metrics are provided as a sample of what can be accomplished with metric options.

### 10.7.1 Class (class)

- **levels of inheritance to most distant base – levels\_to\_base** (count)
 

```
+metric(class.levels_to_base = count(class.immediate_bases) == 0 ? 0 : 1 +
max(class.immediate_bases, class.levels_to_base))
```
- **depth of inheritance tree – depth\_of\_inheritance** (count)
 

```
+metric(class.depth_of_inheritance = count(class.immediate_children) == 0 ? 0 : 1
+ max(class.immediate_children, class.depth_of_inheritance))
```
- **percentage of non-static member functions with public access – public\_member\_function\_percentage** (measure)
 

```
+metric(class.public_member_function_percentage = class.num_public_member_functions
/ class.num_non_static_member_functions)
```
- **weighted methods per class – weighted\_methods\_per\_class** (measure)
 

This metric can be defined in terms of the sum of a chosen member function metric such as cyclomatic complexity or NPATH. For example, one possible choice is: `+metric(class.weighted_methods_per_class = sum(class.non_static_member_functions, function.cyclomatic_complexity))`



## 11 Thread Analysis

### 11.1 Introduction

PC-lint Plus will detect functions used in multiple threads and track mutexes and their lock states when accessing static data from such functions to diagnose a variety of multi-threading issues including:

- unsafe access of static data across multiple threads without the protection of a common mutex.
- calls to thread-unsafe functions.
- inconsistent mutex acquisition order which can lead to deadlocks.
- improper use of trylock functions.
- use of improperly initialized mutexes.
- inappropriate mutex management such as locking a mutex without unlocking it, locking an already locked non-recursive mutex, attempting to unlock a mutex not held by the current thread, or attempting to acquire a shared lock on a mutex in which the current thread already holds an exclusive lock.

Additionally, PC-lint Plus can produce Thread Analysis reports that provide detailed information about functions, threads, mutexes, and/or variables involved in multi-threaded operation.

### 11.2 Library Support

PC-lint Plus contains built-in support for the thread initialization, mutex, and locker functions and classes provided by C11, C++11/14/17, the Qt framework, POSIX pthreads, and the critical section management functions of the Windows synchronization API. Support for other multi-threaded libraries can be configured by employing the Function Semantics feature using the thread semantics described in section [Supporting Other Thread Libraries](#). The `boost.lnt` file (found in the `lnt/` directory of the PC-lint Plus distribution) provides support for Boost's `boost::interprocess` library.

### 11.3 Identifying Threads

Threads in PC-lint Plus are associated with functions, each *thread* having one or more *thread root* functions which are functions from which the thread is started. A thread consists of its root function(s) and all functions called by its root function(s) to any depth. By default, the `main` function is presumed to be a thread root, but options can override that presumption.

While PC-lint Plus supports threads consisting of multiple root functions, in practice threads typically consist of a single root function. In such cases, the name of the thread is the same as the root function unless a different name is specified using the `thread(thread-name)` or `thread_mono(thread-name)` semantics. In the rest of this section, the term *thread* is used interchangeably to refer to both threads and thread root functions while *thread root* is used when it is desired to draw a distinction between the two. Messages that involve threads will reference both the thread name and the root function(s) involved where appropriate.

There are two basic ways of identifying threads: (1) specifying individual functions as thread roots using options and (2) automatically when a function is passed as an argument to a thread creation function.

#### 11.3.1 Options to Identify Threads

The option:

```
-sem( function-name, thread )
```

will identify *function-name* as a thread root. For example,

```
-sem( input_reader, thread )
```

will identify `input_reader` as a thread root function. The name of the function may be fully qualified if a member of a class or namespace.

The option:

```
-sem( function-name, thread( thread-name ) )
```

will identify *function-name* as a thread root of thread *thread-name*. Multiple functions may be specified as thread roots of a thread thereby forming a thread group. Within a thread group, the thread of execution can start in any of the thread root functions and when that function returns the thread of execution may be externally continued at the start of any of the thread root functions in the thread group. Thread groups provide support for Qt QThread slot functions.

By default, it is assumed that a thread can experience multiple concurrent instances. This will trigger the most diagnostics. Consider for example the following code:

```
//lint -sem( f, thread )
void f() {
    static int n = 0;
    n++;
    /* ... */
}
```

This results in:

```
warning 457: variable 'n' is 'written' in function 'f()' of thread 'f' at line 4 of 'test.c'
           which is unprotected with the 'read' in function 'f()' of thread 'f' at line 4 of 'test.c'
warning 457: variable 'n' is 'written' in function 'f()' of thread 'f' at line 4 of 'test.c'
           which is unprotected with the 'write' in function 'f()' of thread 'f' at line 4 of 'test.c'
```

See warning [457](#).

In some cases, however, it is guaranteed that there will be only one instance of a particular thread. We refer to such threads as *mono threads*. The appropriate semantic to provide in such a case is `thread_mono` or `thread_mono(thread-name)`. For example,

```
-sem( function-name, thread_mono )
```

will identify *function-name* as a root function of a mono thread. If `f` had been so identified in the example above, the diagnostic would not have been issued.

The option:

```
-sem( function-name, thread_mono( thread-name ) )
```

will identify *function-name* as a root function of a mono thread *thread-name*.

By default, `main()` would be regarded as a (mono) thread. Removing the thread property from `main()` is accomplished with the `no_thread` semantic:

```
-sem( main, no_thread )
```

### 11.3.2 Automatic Identification of Threads

Using POSIX threads, the identity of a thread can be automatically determined because it is passed as the third argument to the `pthread_create` function. For example:

```
...
pthread_create( &input_thread, NULL, do_input, NULL );
...
```

will allow PC-lint Plus to deduce that `do_input()` is a separate thread.

It is assumed that this thread is not mono. If it is mono and your code depends on that fact, then you will have to identify the thread as `thread_mono` (see above).

Threads created by the Standard C function `thr_create`, the Standard C++ `std::thread` class, and the Standard C++ `std::async` function are similarly automatically recognized. This same recognition of threads can be extended to other functions using the semantic

```
thread_create(i)
```

which identifies the *i*th argument of a function as being a thread-endowing argument. Thus after:

```
-sem( make_thread, thread_create(1) )
```

the call:

```
make_thread( reader, ... );
```

will identify `reader` as a thread.

If PC-lint Plus cannot resolve the function argument of a thread creation function, such as when argument is an array element, thread semantics will not be automatically applied to the function. In this case, the `thread` semantic (see above) can be used to indicate that the function is a thread root.

## 11.4 Mutual Exclusion

To make static analysis meaningful, it is necessary to identify areas of the code in which only a single thread can operate. We will refer to these areas as *thread protected regions*. The use or modification of static variables by two different threads is not considered a violation of thread safety, provided such access lies within a thread protected region. A thread protected region is identified by the acquisition of a mutex to protect against inappropriate concurrent access of shared variables.

PC-lint Plus will automatically recognize calls to POSIX pthread functions `pthread_mutex_lock`, `pthread_mutex_timedlock`, `pthread_mutex_trylock`, `pthread_rwlock_rdlock`, `pthread_rwlock_tryrdlock`, `pthread_rwlock_trywrlock`, and `pthread_rwlock_wrlock` as locking the provided mutex object. Similar support is provided for the Standard C functions `mtx_lock`, `mtx_timedlock`, and `mtx_trylock`, the Standard C++ `std::lock` and `std::try_lock` function templates, the various locking member functions of the `std::mutex`, `std::recursive_mutex`, `std::recursive_timed_mutex`, `std::shared_mutex`, `std::shared_timed_mutex`, and `std::timed_mutex` classes, and the various locking member functions of the Qt mutex classes `QMutex` and `QReadWriteLock`. PC-lint Plus will of course recognize the corresponding mutex unlocking functions as well. Recognition of other mutex locking functions, such as those provided by third-party libraries, can be configured using function semantics (see [Supporting Other Thread Libraries](#)).

In addition to identifying thread protected regions for the purpose of diagnosing unsafe accesses, PC-lint Plus will detect various mutex usage anomalies. For example:

```

#include <mutex>
#include <thread>

std::mutex m1, m2;
int data = 0;

void foo() {
    m1.lock();
    ++data;
    m2.unlock();
}

void bar() {
    std::thread(foo);
}

```

will result in:

```

warning 455: mutex 'm2' unlocked without being locked
    m2.unlock();
    ^
warning 454: mutex 'm1' locked without being unlocked
}
^
supplemental 891: locked here
    m1.lock();
    ^

```

as the mutex being unlocked is not the same as the previously locked mutex. Mutex lock operations are expected to be paired with corresponding unlock operations in the same scope which is generally accepted as good programming practice. Warning 454 is issued when a mutex is not unlocked before the end of the scope in which it was locked. Warning 455 is issued when a mutex is unlocked without having been locked in the same, or enclosing, scope. Warning 456 is issued when two execution paths are combined that do not have the same lock state. For example:

```

//lint -sem(foo, thread)
#include <mutex>

std::mutex m1;
bool g();

void foo() {
    if ( g() ) { m1.lock(); }
    /* ... */
    if ( g() ) { m1.unlock(); }
}

```

will elicit:

```

warning 456: multiple 'if' execution paths are being combined with different lock states
    if ( g() ) { m1.lock(); }
    ^

```

(as well as message 455) while:

```

void foo() {
    if ( g() ) {
        m1.lock();
        /* ... */
    }
}

```

```

        m1.unlock();
    }
}

```

will not.

PC-lint Plus can also detect inconsistent mutex acquisition orders which can result in a deadlock. For example:

```

//lint -sem(foo, thread)
//lint -sem(bar, thread)
#include <mutex>

std::mutex m1, m2;
int data;

void foo() {
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
    ++data;
}

void bar() {
    std::lock_guard<std::mutex> lock2(m2);
    std::lock_guard<std::mutex> lock1(m1);
    ++data;
}

```

will be met with:

```

warning 2462: mutexes 'm1' and 'm2' have lock order mismatch in function 'bar(void)' of
    thread 'bar' at line 16 of 'test.cpp' and function 'foo(void)' of thread 'foo' at
    line 10 of 'test.cpp'
supplemental 891: in function 'bar(void)' of thread 'bar' the lock order is: m2, m1
supplemental 891: in function 'foo(void)' of thread 'foo' the lock order is: m1, m2

```

The inconsistencies will be reported even if they occur across modules.

#### 11.4.1 Trylock-like Functions

Traditional mutex locking functions will block execution of the current thread until the requested mutex is available and only return when the mutex has been successfully acquired. Mutex locking functions that immediately return an error code instead of blocking when the mutex cannot be acquired are referred to as *trylock*-like functions. When calling a *trylock*-like function, it is necessary to examine the return value of the function in order to determine if locking was successful and proceed accordingly.

PC-lint Plus understands the semantics of such functions and the return values indicating successful mutex acquisition individually employed by *trylock*-like functions. Failing to properly utilize the result of a *trylock*-like function will be reported. For example:

```

//lint -sem(foo, thread)
#include <pthread.h>

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
extern int x;

void foo() {
    pthread_mutex_trylock(&m);
}

```

```

    ++x;
    pthread_mutex_unlock(&m);
}

```

will be met with:

```

warning 2511: try_lock return value was discarded
pthread_mutex_trylock(&m);
^

```

along with message 457 to warn about the unsafe access of `x` when mutex `m` is not acquired. The `pthread_mutex_trylock` function returns 0 on success. In the below example, the `x` is mistakenly only updated when this function returns non-zero which will be diagnosed by PC-lint Plus:

```

void foo() {
    if (pthread_mutex_trylock(&m)) {
        ++x;                                // warning 457 - access of 'x' not protected
        pthread_mutex_unlock(&m);           // warning 455 - mutex unlocked without lock
    }
}                                           // warning 454 - mutex locked without unlock

```

The following corrected version will not elicit any such messages and also demonstrates that the result value of a *trylock*-like function may be stored in a variable before being tested.

```

//lint -sem(foo, thread)
#include <pthread.h>

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
extern int x;

void foo() {
    int result = pthread_mutex_trylock(&m);
    if (result == 0) {
        ++x;
        pthread_mutex_unlock(&m);
    }
}

```

If `result` was modified before being tested, warning 2512 would be issued. Note that `pthread_mutex_lock` is also considered to be a *trylock*-like function as it returns a value indicating success or failure.

#### 11.4.2 Locker Classes

In C++, a *locker class* can be used to manage the lock state of one or several mutexes. For example, `std::lock_guard` and `std::scoped_lock` acquire locks for the requested mutexes which are automatically unlocked by the class destructor when the locker goes out of scope and `std::unique_lock` provides a mechanism to manage operations on a set of mutexes.

PC-lint Plus understands the semantics of locker classes and will recognize the lock state changes of the mutexes managed by such locker classes. Additionally, incorrect use of locker classes will be diagnosed. For example:

```

1 //lint -sem(foo, thread)
2 #include <mutex>
3
4 std::mutex m;
5
6 void foo() {
7     m.lock();

```

```

8      /* ... */
9      std::lock_guard g{m};
10     /* ... */
11 }

```

will elicit:

```

warning 2457: non-recursive mutex 'm' locked recursively in function
'foo(void)' of thread 'foo' at line 9 of 'test.cpp'

```

because `std::lock_guard` will attempt to lock `m` which is already locked. `std::lock_guard` can *adopt* a locked mutex by passing a type of `std::adopt_lock_t` in the constructor. PC-lint Plus will similarly warn when an attempt is made to adopt an unowned mutex:

```

#include <mutex>

std::mutex m;

void foo() {
    std::lock_guard g{m, std::adopt_lock};
    /* ... */
}

```

will elicit:

```

warning 2494: mutex 'm' is not locked
std::lock_guard g{m, std::adopt_lock};
                ^
supplemental 891: mutex 'm' ownership taken here
std::lock_guard g{m, std::adopt_lock};
                ^

```

Message [2492](#) will warn when a lock operation is attempted on an already locked locker class. Message [2493](#) will diagnose attempted unlock operations on an unlocked locker class.

### 11.4.3 Shared Locks

Some types of mutexes support *shared* locks and *exclusive* locks, sometimes called *read* and *write* locks. Several threads may hold a *shared* lock on such a mutex but only one *exclusive* lock may be held at a time and no *shared* locks may be held while the mutex is exclusively locked. A *shared* lock creates a *read-only* thread protected region. PC-lint Plus will not complain about *read accesses* to the same static data by multiple threads if each accessing thread holds a shared lock of a common mutex for each access. A *modification* of the same static data that occurs outside an exclusive lock of the same mutex will still be diagnosed.

### 11.4.4 Internal Global Recursive Mutex

The `thread_lock` and `thread_unlock` semantics (see [Supporting Other Thread Libraries](#)) refer to an internal global recursive mutex. This mutex is not user accessible and is named `$Global_Mutex$` in thread analysis messages and reports.

A typical use for these semantics is to apply them to the processor interrupt enable/disable functions of an embedded RTOS.

## 11.5 Function Pointers

If a function has had its address taken, then we presume that we do not know the context of every call made to that function. In the worst case, it could be called by every thread. For that reason, we feel it is meritorious to report on every non-protected access to every static variable by such a function which is done via message [459](#).

## 11.6 Thread Unfriendly Functions

Functions that are inappropriate with some aspects of multi-threaded programs form five categories of severity. This formulation was inspired by severity ratings of hurricanes and, like hurricanes, the higher the number the more severe the function.

- **Category 1:** A function is considered category 1 in a multi-threaded (MT) program if, *when called from more than one thread*, each call needs to be made from a protected region.
- **Category 2:** A function is considered category 2 if it belongs to a group of functions such that whenever two members of this group are called from two different threads, all calls to this group need to be made from a protected region.
- **Category 3:** A function is considered category 3 if *every* call to such a function in an MT program needs to be made from a protected region.
- **Category 4:** A function is considered category 4 if it may only be called from a prescribed subset of the threads.
- **Category 5:** A function is considered category 5 if it may not be called at all.

### 11.6.1 Category 1 Functions

A function is considered thread unsafe if it cannot be used concurrently from two different threads without the protection of a mutex. Generally the function in question is an external function. The reason the function is unsafe is, presumably, that there are static data structures that are manipulated by the function but since the function's source code is not available, it needs to be identified explicitly. However the function does not have to be external. A fully defined function can be indicated as being unsafe and all the cautionary warnings will still be issued.

Functions that are thread unsafe can be identified using the `thread_unsafe` semantic:

```
-sem( function-name, thread_unsafe )
```

Consider the following example:

```
//lint -sem( f, thread_mono )
//lint -sem( g, thread_unsafe )
//lint -sem( h, thread_unsafe )

void g();
void h();

void f() {
    g();
    h();
}

int main() {
    /* ... */
    g();
}
```

Here `main()` and `f()` are threads; functions `g()` and `h()` are both thread unsafe. None of the calls are protected. Warning 460 is issued when it is observed that an unprotected call is made to `g()` from both threads. Warning 460 is not issued for the function `h()` since it is called from only one thread (recall that, by default, `main()` is regarded as a mono thread).



However, info 847 will be issued alerting the user to the fact that thread unsafe functions, `h()` and `g()`, are invoked with unprotected calls. If `h()` is considered Category 3, the call to `h()` should be placed in a critical section. Otherwise the message can be inhibited for this function (or all functions if there are no Category 3 functions in the program).

If `f()` were a thread but not `thread_mono` so that multiple copies of the thread could exist, then warning 460 would be issued for function `h()`.

If either call to `g()` were protected (but not both) warning 460 would still be issued for `g()`. This follows from the principle that no thread is permitted to have unrestricted access to any static data if access is made from two or more threads.

### 11.6.2 Category 2 Functions

An oft-occurring situation is when several external functions in a library access a common pool of data and where thread safety was not part of the library design. Such functions typically can be accessed by one thread even on an unprotected basis without harm. However if another thread accesses any of the functions in the group, then all calls to any member of the group from any thread must be made in a thread protected region. This precisely follows our definition of a Category 2 function.

Category 2 groups of functions are quite common and the usual technique is to designate one thread as the thread that will access the library. If any other thread calls upon the services of the library, such calls will be detected.

To successfully analyze calls to such libraries, the notion of a group of functions is required. The `thread_unsafe` semantic supports an optional group identification. For example, the option:

```
-sem( x, thread_unsafe(xyz) )
```

specifies that `x` is a member of the `xyz` group where `xyz` is an arbitrary name used to designate the group. There are presumably other functions that will use the same group id.

Consider the following example:

```
//lint -sem( f, thread_mono )
//lint -sem( g, thread_mono )
//lint -sem( x, thread_unsafe(xyz) )
//lint -sem( y, thread_unsafe(xyz) )
//lint -sem( z, thread_unsafe(xyz) )

void x();
void y();
void z();

void f() {
    x();
    y();
}

void g() {
    z();
}
```

Here, two threads that are presumed not to have multiple instances of themselves (`thread_mono`) invoke members of a single group (the `xyz` group). Warning 461 is issued for each pair of functions from the `xyz` group that are called from different threads. For example, a warning is issued for the call to function `z()`

because while `z()` is called from a different thread than `x()` and `y()`, it is in the same group as `x()` and `y()` and none of the calls are made from a protected region. Informational messages (847) about the unprotected calls to thread unsafe functions will also be issued.

### 11.6.3 Category 3 Function

To detect abuses of Category 3 functions, info message 847 can be employed. Message 847 will be issued for calls to functions marked with the `thread_unsafe` or `thread_unsafe(group)` semantics when called by a thread, directly or indirectly, outside the protection of a mutex in a multi-threaded program. To cause message 847 to be issued only for specific category 3 functions, the message can be disabled using `-e847` and then enabled for the desired functions using `+esym`. For example:

```
-e847 +esym( 847, cat3func )
```

where `cat3func` is one of the category 3 functions you want to detect. Note that there is no other way to designate a function as Category 3.

### 11.6.4 Category 4 Function

A function may be identified as a Category 4 function by using the semantic `thread_not( list)` or `thread_only( list )` where in each case `list` is a comma separated list of thread names. If a thread appears in a `thread_not` list then that thread may not invoke the function. By contrast, if a thread does not appear in a `thread_only` list then that thread also may not invoke the function. This prohibition from use extends even to thread protected regions.

Consider the following example:

```
//lint -sem( f, thread )
//lint -sem( g, thread )
//lint -sem( a, thread_not(f) )
//lint -sem( b, thread_only( g, main ) )
#include <mutex>

std::mutex m;

void a();
void b();

void c() {
    m.lock();
    b();
    m.unlock();
}

void f() {
    a();
    c();
}
```

Here `f()`, as a thread, invokes (directly or indirectly) `a()`, `b()` and `c()`. A message is issued for `a()` since the semantic `thread_not(f)` explicitly excludes thread `f()`. A message is also issued for `b()` since `b()` may be used only within `g()` or `main()`.

If both the `thread_not` semantic and the `thread_only` semantic are applied to a function, the `thread_not` semantic will take precedence.

Note that the call to `b()` is made from a protected region. Unlike Categories 1 through 3, protecting the call does not eliminate the message.

### 11.6.5 Category 5 Function

Since a function in this category is not to be used at all you could employ the `-deprecate` option. For example, if function `crash` is not to be used at all, the option:

```
-deprecate( function, crash, "MT illegal" )
```

would cause message [586](#) to be issued for each call to the `crash` function.

Alternatively, and perhaps preferably, the `thread_not` semantic may be used. Thus:

```
-sem( crash, thread_not )
```

will have the effect of producing warning [462](#) when `crash` is called. This has the benefit of providing the name of the thread that is invoking `crash`.

## 11.7 Thread Local Storage

Thread Local Storage (TLS) is storage that is allocated separately for each thread initiated. References to TLS data do not need the protection of a mutex and will not be subject to the unsafe access analysis that other static variables are. PC-lint Plus recognizes several syntactic forms for declaring a variable as having thread local storage which are described below.

### 11.7.1 `__Thread_local`

C11 defines the keyword `__Thread_local` which can be used in C modules to declare a TLS variable, e.g.:

```
__Thread_local int n;
```

identifies `n` as being thread local.

### 11.7.2 `thread_local`

C++11 defines the keyword `thread_local` which can be used in C++ modules to declare a TLS variable, e.g.:

```
thread_local int n;
```

identifies `n` as being thread local.

### 11.7.3 `__thread`

The non-standard `__thread` keyword recognized by many compilers is supported by PC-lint Plus and can be used to define TLS variables in both C and C++ modules, .e.g:

```
__thread int n;
```

If your compiler supports a different non-standard keyword to declare TLS variables, the option `-rw_asgn` can be used to cause PC-lint Plus to recognize the TLS-endowing property of the keyword. For example:

```
-rw_asgn(__tls, __thread)
```

will assign the semantics of the `__thread` keyword to the `__tls` keyword which can then be used to declare thread local variables.

### 11.7.4 `__declspec(thread)`

Several popular compilers support the construct `__declspec(thread)` to designate a declaration as TLS. PC-lint Plus supports this behavior when the `fms` flag option is enabled. For example:

```
__declspec(thread) int n;
```

will declare `n` to be thread local.

## 11.8 Limitations

- Mutex lock and unlock functions receiving a mutex pointer or reference for which PC-lint Plus cannot deduce the underlying mutex will not be recognized, which may result in false positive messages. This can occur when locking a mutex from an array of mutex objects (PC-lint Plus does not currently track array elements) or if a function locks a mutex that is passed to it as an argument. Message [2751](#)(indeterminable mutex) will be issued in such cases.
- Mutex lock and unlock operations are expected to be properly paired within the same scope, violations of this expectation are reported by messages [454](#), [455](#), and [456](#).
- Thread operations (such as `join`) that affect the lifetime of a thread are not currently recognized by PC-lint Plus which may result in safe accesses that occur before or after the lifetime of a thread being incorrectly reported as unsafe.
- An access of an array element or structure member is considered an access of the entire array/structure. If e.g. two threads access different members of the same structure or class outside the protection of a common mutex, this will be reported as an unsafe access.
- Variables passed by pointer/reference as an argument in a function call will be assumed to be accessed at the point of the call.
- Modifications made to static data through indirect accesses which cannot be resolved by PC-lint Plus will not be reported as unsafe when occurring outside the protection of a mutex.

## 11.9 Thread Analysis Reports

PC-lint Plus can generate several types of thread analysis reports which can provide additional insight about an analyzed program's use of threads, functions, mutexes, and shared variables. These reports can be emitted in XML, JSON, CSV, or plain text formats. The types of reports available and the data they contain are:

- Thread report - the threads identified during thread analysis and their root functions.
- Function report - functions called by threads, the shared variables they access and the type of access (i.e. read/write), the mutexes held at each access, and calls made by the function (directly or indirectly).
- Mutex report - mutexes used, their type (i.e. shared/recursive), and the locations of their use.
- Variable report - shared variables accessed by threads.

Reports are requested using the `-thread_report` option which accepts three sub-options and a number of flags. The sub-options are:

- `type=report-type` where *report-type* is one of `threads`, `functions`, `mutexes`, or `variables`.
- `file=output-file` where *output-file* is the name of the file to write the report to. If the specified file already exists, it is overwritten.
- `format=output-format` where *output-format* is one of `csv`, `json`, `text`, or `xml`.

The `type` and `file` sub-options are required. The `format` option defaults to `text` if not provided.

Flags are either report *filters* or *fields*. *Filter* flags affect which entries are included in the report. *Field* flags are used to override the default set of fields that will be present in the report. For example:

```
-thread_report(type=functions, file=funcs.txt, function,variables)
```

will produce a function report containing the **function** and **variables** fields. Specifying a field that contains children (such as **variables** above) will automatically include the children as well.

Each type of report supports a unique set of *filter* flags that are described in the individual reports below.

### 11.9.1 Field Types and Representation

Each field in a report has a static value type which is one of Boolean, Numeric, String, Location, Array, Tristate, or an aggregate combination of one or more of these types. An Array is a list of zero or more items of a single type. A Location consists of a file name and a line number. The representation of values of each type is dependent on the output format selected. Boolean types are represented as **true** or **false** in JSON, **1** or **0** in XML and CSV, and **yes** or **no** in plain text format. Tristate values may be **true**, **false**, or **unknown**. Numeric values are integers represented using base-10 digits in all formats. Strings are surrounded by double quotes in JSON, XML, and CSV.

In JSON, a Location is represented as an object with a **file** and **line** keys. In XML, **location** is an element with **line** and **file** attributes. A Location in CSV is represented with the line number and file name in a single quoted field, separated by a comma. In a plain text report, a Location consists of a line number and a double-quoted file name separated by a space.

### 11.9.2 Thread Reports

A *thread* report includes information about the threads identified in the program, the root function(s) associated with each thread, and whether the thread is a mono thread. By default, all threads are included in the report. The **error** filter flag may be specified in the **thread\_report** option to indicate that only threads referenced by messages [457](#), [460](#), [461](#), [462](#), [847](#), [2457](#), or [2462](#) should be included in the report. The available fields for the thread report are:

Field	Type	Description
<b>thread</b>	String	The name of the thread
<b>mono</b>	Boolean	Indicates whether the thread has <b>thread_mono</b> semantics
<b>in_error</b>	Boolean	Indicates whether the thread is referenced in messages <a href="#">457</a> , <a href="#">460</a> , <a href="#">461</a> , <a href="#">462</a> , <a href="#">847</a> , <a href="#">2457</a> , or <a href="#">2462</a>
<b>roots</b>	Array	The thread root function(s)
↳ <b>root</b>	String	The name of the root function
↳ <b>definition</b>	Location	The definition location of the root function

### 11.9.3 Function Reports

A *function* report includes information about the functions that participate in thread analysis, i.e. thread root functions and the functions they call directly or indirectly. This information includes thread semantics (such as **thread\_protected** or **thread\_unsafe**), variables accessed by the function, the type and location of those accesses, and the locks held for each access, the functions called by each function, and the lock state and location of each call. The **all** filter flag may be used to request that *all* functions be included in the report, not just those related to thread operations.

Field	Type	Description
<b>function</b>	String	The function name and parameter list
<b>definition</b>	Location	The definition location of the function.
<b>protected</b>	Boolean	Indicates whether the thread has <b>thread_protected</b> semantics
<b>thread_not</b>	Boolean	Indicates whether the thread has <b>thread_not</b> semantics
<b>thread_unsafe</b>	Boolean	Indicates whether the thread has <b>thread_unsafe</b> semantics
<b>addressed</b>	Boolean	Indicates whether the function's address was taken
<b>in_analysis</b>	Boolean	Whether the function is included in the thread analysis
<b>variables</b>	Array	Array of variables directly accessed by the function. Each item in the array is an aggregate containing the following members:
↳ <b>name</b>	String	The name of the variable accessed by the function
↳ <b>accesses</b>	Array	Array of accesses of the variable. Each item in the array is an aggregate containing the following members:
↳ <b>type</b>	String	The type of access, one of <b>read</b> or <b>write</b>
↳ <b>locked</b>	String	A comma-separated list of mutexes locked by the function at the access point
↳ <b>access_locations</b>	Array	An array of locations where the variable is accessed with the associated access and mutexes held
<b>calls</b>	Array	Array of functions called by <b>function</b> . Each item in the array is an aggregate containing the following members:
↳ <b>called</b>	String	The called function
↳ <b>states</b>	Array	Array of function call information. Each item in the array is an aggregate containing:
↳ <b>shared</b>	String	A comma-separated list of mutexes locked for reading in the order they were locked
↳ <b>exclusive</b>	String	A comma-separated list of mutexes locked for writing in the order they were locked
↳ <b>call_locations</b>	Array	An array of locations where the function was called with the associated lock states

By default, only the first location of each called function with a unique mutex lock ordering is reported. Enabling the **faf** flag option (**+faf**) will cause all function call locations to be included in the report. Similarly, only the first location of each variable access with a unique access type and set of owned mutexes is included in the report by default. If the **fav** flag option is enabled (**+fav**), all accesses to each variable will be reported. The **faf** and **fav** flag options may be enabled for specific source code regions (e.g. by surrounding a region with lint comments containing **++faf/++fav** and **--faf/--fav**) to enable reporting of all calls and/or variable accesses within the corresponding regions.

#### 11.9.4 Mutex Reports

A *mutex* report contains information about the mutexes used in the program, their type, and the locations of their initializations and references. The available filter flags for this report are: **2530**, **2531**, **2770**, and **2771**. When one or more of these flags are provided, the report is limited to those mutexes that were referenced by the corresponding messages of the same number.

Field	Type	Description
<b>name</b>	String	The name of the mutex
<b>recursive</b>	Tristate	Whether the mutex was initialized as a recursive mutex
<b>shared</b>	Tristate	Whether the mutex was initialized as a shared mutex
<b>effective_recursive</b>	Boolean	Whether the mutex was deduced as being recursive from its use
<b>effective_shared</b>	Boolean	Whether the mutex was deduced as being shared from its use
<b>in_2530</b>	Boolean	Whether the mutex is referenced in message <b>2530</b>

<code>in_2531</code>	Boolean	Whether the mutex is referenced in message <a href="#">2531</a>
<code>in_2770</code>	Boolean	Whether the mutex is referenced in message <a href="#">2770</a>
<code>in_2771</code>	Boolean	Whether the mutex is referenced in message <a href="#">2771</a>
<code>init_locations</code>	Array	An array of locations where the mutex was initialized
<code>ref_locations</code>	Array	An array of locations where the mutex was directly referenced

### 11.9.5 Variable Reports

The *variable* report provides the set of variables collected during thread analysis. The `all` filter flag may be used to request *all* variables be included in the report, not just those related to thread operations.

Field	Type	Description
<code>variable</code>	String	The name of the variable
<code>in_analysis</code>	Boolean	Whether the variable is included in the thread analysis
<code>definition</code>	Location	The location of the variable's definition

## 11.10 Message Summary

The below table summarizes the messages supporting the Thread Analysis feature. The phase indicates the thread analysis phase at which the message is issued (see "Thread Analysis Phases" below).

Message #	Phase	Summary Description
<a href="#">454</a>	module	mutex locked without being unlocked
<a href="#">455</a>	module	unlock without being locked
<a href="#">456</a>	module	multiple execution paths are being combined with different lock states
<a href="#">457</a>	global	function of thread has access to variable which is unprotected with the access by function of thread
<a href="#">459</a>	global	function whose address was taken has an unprotected access of variable
<a href="#">460</a>	global	thread has unprotected call to thread unsafe function which is also called by thread
<a href="#">461</a>	global	thread has unprotected call to function of group while thread calls function of the same group
<a href="#">462</a>	global	thread calling function is inconsistent with the semantic
<a href="#">847</a>	global	thread has unprotected call to thread unsafe function
<a href="#">2457</a>	global	non-recursive mutex locked recursively in thread
<a href="#">2462</a>	global	mutex lock order mismatch
<a href="#">2463</a>	module	statement while locked
<a href="#">2467</a>	module	multiple definitions of function
<a href="#">2468</a>	module	semantic mismatch for function
<a href="#">2482</a>	module	enumeration constant not defined
<a href="#">2483</a>	module	invalid semantics for function
<a href="#">2484</a>	module	invalid semantics for argument
<a href="#">2485</a>	module	invalid semantics for locker class
<a href="#">2486</a>	module	invalid semantics for function
<a href="#">2488</a>	module	mutex passed more than once
<a href="#">2489</a>	module	share lock/unlock an exclusively locked mutex/locker
<a href="#">2490</a>	module	symbolic constant not defined
<a href="#">2492</a>	module	locker already locked
<a href="#">2493</a>	module	locker not locked

---

2494	module	locked mutex required
2495	module	mutex not locked/unlocked
2496	module	locker not locked/unlocked
2511	module	try_lock return value was discarded
2512	module	try_lock return value was modified
2513	module	try_lock return value was manipulated
2520	module	mutex_attr is not initialized
2521	module	mutex_attr already destroyed
2522	module	mutex_attr already initialized
2530	global	mutexes has inconsistent types
2531	global	mutex has usage incompatible with its type
2751	module	indeterminable ...
2752	module	incompatible semantics for locker class, corrections applied
2753	module	mutex may not be locked/unlocked
2770	global	type of mutex is incomplete but its usage requires specific type
2771	global	type of mutex is unknown
2920	module	unlocking order mismatch

---

## 11.11 Thread Analysis Phases

Thread-related analysis occurs in two distinct phases. The first phase occurs during module analysis where module-level threading issues that can be detected during this phase are reported. The second phase occurs after Global Wrap-up and uses the combined information collected during the individual module processing phases to model the inter-module relationships necessary to detect issues from a holistic perspective.

Messages [457](#), [459](#), [460](#), [461](#), [462](#), [847](#), [2457](#), [2462](#), [2530](#), [2531](#), [2770](#), and [2771](#) are issued in the inter-module (second) phase, with the remaining thread analysis messages issued during module analysis. The messages issued during the inter-module phase do not contain the file and line number information that typically accompanies messages and are not subject to location-based suppressions such as one-line suppressions, next-expression or next-statement suppressions, or `-efile` suppressions. Relevant location information is instead provided in the text of the message and/or within the text of supplemental messages. Other suppressions such as `-esym`, `-estring`, `-etype`, `-egrep`, and `-e` are honored for these messages but must be active at the point these messages are issued to be effective. In other words, to use e.g. `-esym` for message [457](#), the option must appear outside of a module and still be in effect after all modules have been processed (the lifetime of options encountered within a module are limited to the module in which they appear).

### 11.11.1 Inhibition of Thread Analysis

Complete inhibition of thread analysis can be accomplished by turning off the `ftc` flag option with `-ftc`. While thread analysis is disabled, no thread-related messages will be reported and the inter-module thread analysis phase described above will not occur. There are several situations outside the use of `-ftc` in which the inter-module phase of thread analysis will not be performed. These situations are:

- When Global Wrapup is suppressed with `-unit_check` or `-u` options. Inter-module thread analysis occurs after Global Wrapup and will not be performed if Global Wrapup does not occur.
- When no threads or mutexes are detected during module analysis. The checks performed by thread analysis in the inter-module phase are not relevant for programs that do not utilize threads or mutexes.
- After the attempted issuance of messages [2467](#) or [2468](#). If PC-lint Plus attempts to issue one of these messages but the message is suppressed, inter-module thread analysis will still be disabled. This condition can be overridden by setting the `ftc` flag option to 2 or greater.

## 11.12 Supporting Other Thread Libraries

The builtin support that PC-lint Plus provides for the Qt, POSIX, and standard C and C++ thread libraries can be extended to other thread libraries by employing [Function Semantics](#) to designate functions that create



and operate on threads, mutexes, and locker classes. PC-lint Plus provides a rich set of *semantics* for this purpose. The following sections detail these semantics.

Note that many of the semantics must be used in conjunction with other semantics. For example, the `mutex_destroy` semantic must be used with the `mutex` semantic to indicate which mutex argument is destroyed. When multiple semantics are used with the same function, all of these semantics should be included within the same `-sem` option, e.g.:

```
-sem( foo, mutex_destroy, mutex(1) )
```

and not:

```
-sem( foo, mutex_destroy )
-sem( foo, mutex(1) )
```

PC-lint Plus performs semantic validation of thread semantics as each `-sem` option is processed and will produce an error if a `-sem` option does not include dependent semantics.

### 11.12.1 Thread Semantics

#### **async**

Indicates that the function is comparable to the standard C++ `std::async` function template.

#### **thread**

Indicates that the function is a thread root. See [Options to Identify Threads](#) for additional information.

#### **thread(thread\_name)**

Indicates that the function is a thread root of thread *thread\_name*. This semantic takes precedence over the `thread` semantic. See [Options to Identify Threads](#) for additional information.

#### **thread\_mono**

Indicates that the function is a mono thread root. This semantic takes precedence over the `thread` semantic. See [Options to Identify Threads](#) for additional information.

#### **thread\_mono(thread\_name)**

Indicates that the function is a mono thread root of thread *thread\_name*. This semantic takes precedence over the `thread_mono` semantic. See [Options to Identify Threads](#) for additional information.

#### **thread\_args(n)**

Indicates that the *n*th argument and all following arguments should be passed as arguments to the associated thread root function. Requires the `thread_create(n)` function semantic with a value for *n* less than the value of the *n* of the `thread_args(n)` argument semantic.

#### **thread\_atomic(n)**

Indicates that the *n*th parameter is a pointer or reference to an object which will only be accessed atomically within the function. By default, PC-lint Plus assumes that a pointer or reference parameter to `non-const` will be used to access the target object for non-atomic writes and a pointer or reference to `const` will be used to access the target for non-atomic reads. If the function parameter is declared as pointer or reference to an atomic type, atomic access will be assumed. This semantic is useful for specifying that a parameter not declared as pointer or reference to atomic will only be accessed atomically.

This semantic may not be combined with any other mutex argument semantics or the `mutex_validate` function semantic. This semantic may not be combined with a `mutex_remaining(n)` or `thread_args(n)`

argument semantic with a value of *n* that is less than the *n* used in this semantic.

#### **thread\_create(*n*)**

Indicates that the *n*th argument of the function is the address of a thread root function.

#### **thread\_immune(*n*)**

Indicates that the *n*th argument is thread immune and will not participate in thread analysis. This argument semantic prevents the 'read access' or 'write access' of the corresponding variable from being tracked and possibly being reported in thread analysis as being unprotected with the access within another thread.

#### **thread\_non\_atomic(*n*)**

Indicates that the *n*th parameter is a pointer or reference to an object that may be accessed non-atomically. By default, PC-lint Plus assumes that a pointer or reference parameter will be used to access the target object non-atomically but if the target is an atomic type, atomic access is assumed. This semantic will cause PC-lint Plus to assume non-atomic access to the target, even if the target is an atomic type. For example, the C++ `std::atomic_init` function template takes a pointer to an atomic type as the first argument but instantiations of this function may access the target non-atomically.

This semantic may not be combined with any other mutex argument semantics or the `mutex_validate` function semantic. This semantic may not be combined with a `mutex_remaining(n)` or `thread_args(n)` argument semantic with a value of *n* that is less than the *n* used in this semantic.

#### **thread\_protected**

Indicates that the entire function is a thread-protected region as if it were protected by an invisible, function-specific, recursive mutex. Useful for specifying functions where the compiler provides guarantees against concurrent execution using extensions not recognized by PC-lint Plus.

### 11.12.2 Mutex Semantics

#### **mutex(*n*)**

Indicates that the *n*th argument is a pointer or reference to a mutex. This semantic is used with the `mutex_destroy`, `mutex_initialize`, `mutex_lock`, `mutex_lock_shared`, `mutex_unlock`, and `mutex_unlock_shared` semantics to specify the mutex argument being destroyed, initialized, locked, or unlocked. The mutex semantic may also be combined with at most one of `mutex_attribute`, `mutex_is_recursive`, or `mutex_is_shared`. A value of *t* may be used for *n* to specify the `this` object instead of an explicitly passed argument.

#### **mutex\_attribute(*n*)**

When combined with the `mutex(n)` argument semantic, indicates that the *n*th argument is a plain (non-recursive and non-shared) mutex.

Otherwise, this semantic indicates that the *n*th argument is a pointer or reference to a mutex attribute structure (e.g. `pthread_mutexattr_t` or `pthread_rwlockattr_t`). This usage of the semantic is only valid if used with the `mutex_attribute_destroy`, `mutex_attribute_initialize`, or `mutex_attribute_set` mutex special semantics.

This argument semantic may be combined with the `mutex_is_recursive(n)` and/or `mutex_is_shared(n)` argument semantics if the function has either the `mutex_attribute_initialize` or the `mutex_attribute_set` mutex semantics.

#### **mutex\_attribute\_destroy**

Indicates that the function destroys the mutex attribute object received as an argument. This semantic must be combined with a `mutex_attribute(n)` semantic where *n* is the attribute argument that will be

destroyed by the function.

#### **mutex\_attribute\_initialize**

Specifies that the function initializes the mutex attribute object received as an argument. This semantic must be combined with a `mutex_attribute(n)` semantic where *n* is the attribute argument that will be initialized by the function. The `mutex_is_recursive` and/or `mutex_is_shared` argument semantics may be used to specify the default settings of the mutex attributes object.

#### **mutex\_attribute\_set**

Specifies that the function sets the attributes of a mutex attribute object. This semantic must be combined with a `mutex_attribute(n)` semantic where *n* is the attribute argument that will be set by the function.

#### **mutex\_destroy**

Indicates that the function will destroy a mutex. This semantic must be combined with a `mutex(n)` semantic where *n* is the mutex argument to be destroyed.

#### **mutex\_ignore**

Specifies that the function will not be walked during value tracking. Useful for functions which accept one or more mutex, mutex attribute, or mutex locker arguments but do not modify the state of those objects to prevent PC-lint Plus from making inferences about them while processing calls to the function.

#### **mutex\_initialize**

Specifies that the function initializes a mutex argument. This semantic must be combined with a `mutex(n)` argument semantic which specifies the mutex argument initialized as well as at least one of `mutex_is_recursive(n)`, `mutex_is_shared(n)`, or `mutex_attribute(n)` to indicate that a mutex initialized with this function will be recursive, shared, or plain, respectively.

#### **mutex\_initialize\_std\_c**

Specifies that the function initializes a mutex argument. This semantic must be combined with a `mutex(n)` argument semantic which specifies the mutex argument initialized as well as at least one of `mutex_is_recursive(n)`, `mutex_is_shared(n)`, or `mutex_attribute(n)` to indicate that a mutex initialized with this function will be recursive, shared, or plain, respectively.

The `mutex_is_recursive(n)` argument's value will be ANDed with the value of the `mtx_recursive` enumeration constant to determine if the mutex is a recursive mutex. If the `mtx_recursive` enumeration constant has not been defined when a function with this special semantic is called, warning 2482 will be emitted.

#### **mutex\_is\_recursive(*n*)**

When combined with the `mutex(n)` argument semantic, indicates that the *n*th argument is a recursive mutex. When combined with the `mutex_attribute(n)` argument semantics (only if the function has either the `mutex_attribute_initialize` or `mutex_attribute_set` mutex semantics), indicates that the mutex attribute argument will specify a recursive mutex (see [-mutex\\_attr](#) for details of this behavior). Otherwise, indicates that the *n*th argument controls whether or not the mutex is recursive.

#### **mutex\_is\_shared(*n*)**

When combined with the `mutex(n)` argument semantic, indicates that the *n*th argument is a shared mutex. When combined with the `mutex_locker` argument semantic, indicates that the *n*th argument is a shared locker. When combined with the `mutex_attribute(n)` argument semantics (only if the function has either the `mutex_attribute_initialize` or `mutex_attribute_set` mutex semantics), indicates that the mutex attribute argument will specify a shared mutex (see [-mutex\\_attr](#) for details of this behavior). Otherwise, indicates that the *n*th argument controls whether or not the mutex is shared.

#### **mutex\_lock**

The specified function exclusively locks a mutex. This semantic must be accompanied by one or more `mutex(n)` semantics or exactly one `mutex_remaining(n)` semantic which specify the argument(s) locked by the function. If the return type of the function is not `void`, the function must be specified with a try-lock return semantic other than `try_lock_none`.

#### **mutex\_lock\_shared**

The specified function locks a mutex in shared mode. This semantic must be accompanied by one or more `mutex(n)` semantics or exactly one `mutex_remaining(n)` semantic which specify the argument(s) locked by the function. If the return type of the function is not `void`, the function must be specified with a try-lock return semantic other than `try_lock_none`.

#### **mutex\_must\_be\_locked(*n*)**

This semantic must be used with `mutex_validate`. Indicates that the mutex object received as argument *n* must be locked. Supports functions that expect to receive a locked mutex argument such as `std::condition_variable::wait`.

#### **mutex\_must\_be\_unlocked(*n*)**

This semantic must be used with `mutex_validate`. Indicates that the mutex object received as argument *n* must be unlocked. Supports functions that expect to receive an unlocked mutex argument.

#### **mutex\_remaining(*n*)**

Like the `mutex` semantic, indicates that the *n*th argument is a pointer or reference to a mutex but also that all following arguments are pointers or references to mutexes as well. The mutex arguments are also checked for being null. *n* is not allowed to be `t`. This semantic may be used with any combination of `mutex_tag_adopt`, `mutex_tag_defer`, and `mutex_tag_try_to_lock` to indicate that the last argument may be corresponding locker tag type.

#### **mutex\_tag\_adopt(*n*)**

When used with the `mutex_remaining` argument semantic, indicates that the last argument may be a `std::adopt_lock_t` tag type. Otherwise, indicates that the *n*th argument may be a `std::adopt_lock_t` tag type. *n* may not be `t`. This semantic may also be combined with `mutex_tag_defer` and/or `mutex_tag_try_to_lock`. Alternate tag types may be specified with the `-locker_tag` option.

#### **mutex\_tag\_defer(*n*)**

When used with the `mutex_remaining` argument semantic, indicates that the last argument may be a `std::defer_lock_t` tag type. Otherwise, indicates that the *n*th argument may be a `std::defer_lock_t` tag type. *n* may not be `t`. This semantic may also be combined with `mutex_tag_adopt` and/or `mutex_tag_try_to_lock`. Alternate tag types may be specified with the `-locker_tag` option.

#### **mutex\_tag\_try\_to\_lock(*n*)**

When used with the `mutex_remaining` argument semantic, indicates that the last argument may be a `std::try_to_lock_t` tag type. Otherwise, indicates that the *n*th argument may be a `std::try_to_lock_t` tag type. *n* may not be `t`. This semantic may also be combined with `mutex_tag_adopt` and/or `mutex_tag_defer`. Alternate tag types may be specified with the `-locker_tag` option.

#### **mutex\_unlock**

The function unlocks a mutex. This semantic must be accompanied by exactly one `mutex(n)` semantic specifying the mutex argument which will be unlocked.

#### **mutex\_unlock\_shared**

The function unlocks a share-locked mutex. This semantic must be accompanied by exactly one `mutex(n)` semantic specifying the mutex argument which will be unlocked.

#### **mutex\_validate**

This semantic must be used with one or more of `mutex`, `mutex_attribute`, or `mutex_locker` to specify the arguments that should be validated. Implies the semantics of `mutex_ignore` and requests validation of the specified mutex(es), attribute(s), and/or locker objects during calls to the target function. The `mutex_must_be_locked` or `mutex_must_be_unlocked` argument semantics may be combined with `mutex_validate` to ensure that the provided mutex or mutex locker argument is locked or unlocked when the function is called (violations will be reported with messages [2753](#), [2495](#), and [2496](#)).

#### **thread\_lock**

The specified function exclusively locks the internal global recursive mutex `mutex`.

#### **thread\_unlock**

The specified function unlocks the internal global recursive mutex `mutex`.

### **11.12.3 Locker Semantics**

A *locker class* is a class which manages the lock states of one or more mutex objects. Examples of locker classes in the standard C++ library include `std::lock_guard` and `std::scoped_lock`. The operations performed by such classes can be configured using the semantics described in this section.

Locker classes whose member functions are targeted by the semantics described in this section must confirm to the following requirements:

- At least one non-move, non-copy constructor must be endowed with the `locker_create` semantic.
- A non-deleted destructor must be provided.
- No copy constructors or copy assignment operators may be defined.
- The use of the `locker_move`, `locker_assign`, and `locker_release` semantics may only be used if the locker class defines a non-deleted default constructor.

#### **locker\_assign**

This semantic must be used in conjunction with two `mutex_locker` argument semantics. The semantic specifies that mutexes owned by the second `mutex_locker` argument are moved to the first `mutex_locker` argument. Any mutexes previously owned by the first `mutex_locker` argument are released. Supports move assignment operations for classes like `std::unique_lock`.

#### **locker\_create**

Indicates the specified function creates a locker object. This semantic implicitly sets `mutex_locker(t)`. The `mutex_is_shared(t)` argument semantic can be used to indicate the locker object supports shared mutexes. This semantic must be accompanied by one or more `mutex(n)` semantics or exactly one `mutex_remaining(n)` semantic. This semantic may be used with any combination of `mutex_tag_adopt`, `mutex_tag_defer`, and `mutex_tag_try_to_lock` to indicate that the last argument of the locker creation function may be a corresponding locker tag type.

#### **locker\_destroy**

Indicates that the function destroys a locker object. This semantic implicitly sets the `mutex_locker(t)` semantic.

#### **locker\_fetch**

This semantic must be used in conjunction with exactly one `mutex_locker` argument semantic. The function returns the mutex tracked by the locker object. Supports mutex-fetching functions such as `std::unique_lock::mutex`.

#### **locker\_lock**

The specified locker object argument will exclusively lock its mutex(es). This semantic must be combined with exactly one `mutex_locker(n)` semantic which specifies the argument corresponding to the locker object on which the lock operation will be performed. If the return type of the function is not `void`, the function must be specified with a try-lock return semantic other than `try_lock_none`.

#### **locker\_lock\_shared**

The specified locker object argument will share-lock its mutex(es). This semantic must be combined with exactly one `mutex_locker(n)` semantic which specifies the argument corresponding to the locker object on which the lock operation will be performed. If the return type of the function is not `void`, the function must be specified with a try-lock return semantic other than `try_lock_none`.

#### **locker\_move**

This semantic must be used in conjunction with two `mutex_locker` argument semantics. The specified function creates a new locker object and the mutexes owned by the second `mutex_locker` argument are moved to the first `mutex_locker` argument. Supports move construction for classes like `std::unique_lock`.

#### **locker\_owns**

This semantic must be used in conjunction with exactly one `mutex_locker` argument semantic. The function returns `true` if the locker argument specified by the `mutex_locker` semantic owns a locked mutex and returns `false` otherwise. This semantic implicitly sets the `try_lock_true` semantic for the function.

#### **locker\_release**

This semantic must be used in conjunction with exactly one `mutex_locker` argument semantic. The function releases and returns the mutex owned by the locker argument specified by `mutex_locker` without unlocking it. Supports mutex-releasing functions such as `std::unique_lock::release`.

#### **locker\_swap**

This semantic must be used in conjunction with two `mutex_locker` argument semantics. The function swaps the mutex(es) and ownership status of the two locker arguments. Supports swap semantics for functions like `std::unique_lock::swap`.

#### **locker\_unlock**

The mutex(es) owned by the specified locker argument are unlocked. This semantic must be combined with a single `mutex_locker(n)` semantic which specifies the locker argument operated on.

#### **mutex\_locker(*n*)**

This semantic is used to indicate that the specified argument is a pointer or reference to a mutex locker object. `mutex_locker` is used in conjunction with `locker_assign`, `locker_fetch`, `locker_lock`, `locker_lock_shared`, `locker_move`, `locker_owns`, `locker_release`, `locker_swap`, and `locker_unlock` to indicate the locker object(s) involved in the corresponding operation. A value of `t` may be used for `n` to indicate that the mutex locker object acted on is the `this` object (a value of `t` is required when used with `locker_create` and `locker_destroy`).

### **11.12.4 Trylock Semantics**

A *trylock* function is a mutex locking function that may fail and whose success is communicated through its return value. The return values indicating success or failure may be specified for individual trylock functions using the semantics described in this section. Functions that are the target of a `mutex_lock`, `mutex_lock_shared`, `locker_lock`, or `locker_lock_shared` semantic that do not specify a return value of `void` must be specified with one of the below trylock semantics other than `try_lock_none`.

#### **try\_lock\_false**

Indicates that the associated function returns `false` on success.

**try\_lock\_neg\_1**

Indicates that the associated function returns -1 (negative one) on success.

**try\_lock\_none**

Indicates that the associated function does not return anything indicating success. This is the default if no other try lock return semantic is provided.

**try\_lock\_one**

Indicates that the associated function returns 1 (one) on success.

**try\_lock\_std\_c**

Indicates that the associated function returns the value of the `thrd_success` enumeration constant on success. A warning message is emitted if the definition of the `thrd_success` enumeration constant has not been seen prior to the call of a function with this semantic.

**try\_lock\_true**

Indicates that the associated function returns `true` on success.

**try\_lock\_zero**

Indicates that the associated function returns 0 (zero) on success.

## 12 MISRA<sup>®</sup> Standards Checking

The Motor Industry Software Reliability Association (MISRA) is an organization that produces and maintains C and C++ programming guidelines. The primary purpose of these guidelines is to codify a set of recommendations related to software development that aids in the creation of "safe and reliable software". While MISRA is an effort born out of the automotive industry, MISRA's success has grown and the guidelines have been adopted to meet needs in other safety-critical industries such as healthcare and aerospace.

MISRA has produced three versions of their guidelines for C, each one replacing the previous version. The versions are MISRA C 1998 (sometimes referred to as MISRA C1), MISRA C 2004 (aka MISRA C2) and MISRA C 2012 (aka MISRA C3). In 2008, MISRA released guidelines for C++ (MISRA C++). In 2016 MISRA released an amendment to MISRA C 2012 (MISRA C 2012 AMD-1) which adds several new rules. In 2020 MISRA released a second amendment to MISRA C 2012 (AMD-2) adding two new rules and updating several others. A third amendment to MISRA C 2012 (AMD-3) was released in 2022 and a fourth amendment (AMD-4) in 2023. While the MISRA C++ effort is currently defunct (there is no active work in this area), the guidelines are employed by some organizations seeking MISRA style guidelines for C++. The AUTOSAR coding guidelines (also supported by PC-lint Plus) is essentially a modern version of MISRA C++.

Each MISRA guidelines document consists of a series of numbered advisory, required, and mandatory "rules" and "directives". A directive is more generalized (such as requiring that "run-time failures be minimized") while Rules are concrete and testable (such as forbidding the use of C++ style comments). Directives are often not statically checkable while Rules often are.

PC-lint Plus provides support for the MISRA C2, MISRA C3 (including AMD-1, AMD-2, AMD-3, and AMD-4), and MISRA C++ guidelines. This support is achieved through a combination of standard PC-lint Plus messages and elective notes dedicated to specific MISRA rules. Vector Informatik GmbH provides the author files `au-misra2.lnt`, `au-misra3.lnt`, `au-misra3-amd1.lnt`, `au-misra3-amd2.lnt`, `au-misra3-amd3.lnt`, `au-misra3-amd4.lnt`, and `au-misra-cpp.lnt` to enable the checks necessary to support these guidelines. These author files include `-append` options which cause messages that are used to report on MISRA violations to be annotated with the corresponding Rule or Directive number(s).

While some of the messages are very specific to MISRA guidelines (such as those involving interactions amongst "essential types", a MISRA creation), any of the messages may be employed individually for those desiring to make use of a subset of the checks, outside of MISRA compliance checking.

**The author files enable checks for both library and non-library code. This means that the standard headers used in your source code are subject to the same scrutiny as the rest of the project. This is often a requirement but can result in a lot of noise if library code is not subject to the same compliance requirements as the rest of the project. MISRA checks can be disabled for library code by placing the options `-wlib(4)` `-wlib(1)` immediately after the author file is referenced. This raises and immediately lowers the library warning level resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.**

The following subsections document the level of support provided by PC-lint Plus for each of the corresponding guidelines. For each guideline, the guideline name, level of support, and primary enforcing messages are provided. The letters within parentheses after each rule indicate whether the rule is advisory, required, documentation, or mandatory. An asterisk inside the parentheses indicates that MISRA has deemed the rule to be "undecidable", that is not possible to be fully checked by static analysis methods. In such cases PC-lint Plus provides the level of support feasible. While every effort is made to ensure the correctness of the information provided here, Vector Informatik GmbH makes no guarantee regarding the accuracy of the information conveyed.



The following terms are used to characterize the support that PC-lint Plus provides each guideline:

- **Supported** - For statically checkable guidelines, the rule is comprehensively supported and no false positives nor false negatives are expected. For guidelines that are not fully statically checkable, substantial support is provided to detect statically checkable violations.
- **Partially Supported** - Meaningful support is provided but there may be cases where false positives and/or false negatives may occur due to limitations in the currently implemented detection method, one or more guideline exceptions are not implemented, etc.
- **Assistance Provided** - While enforcement of the actual guideline is not supported (or cannot be statically checked), *potential* violations of the guideline are diagnosed.
- **Not Supported** - No meaningful level of support is currently provided for this guideline.
- **Not Statically Checkable** - Violations of the guideline cannot be detected by means of static analysis.

Most guidelines marked as *Partially Supported* or *Assistance Provided* contain a footnote that corresponds to an explanation of the corresponding limitation(s) at the end of each support matrix.

---

<sup>®</sup> MISRA is a registered trademark of HORIBA MIRA Ltd, held on behalf of the MISRA Consortium.

## 12.1 MISRA C 2012

### 12.1.1 MISRA C 2012 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	145	94.2%
Comprehensive	136	88.3%
Partial	7	4.5%
Assistance	2	1.3%
Not Supported	9	5.8%
Not Statically Checkable	5	
TOTAL	159	

### 12.1.2 MISRA C 2012 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Dir 1.1 (Req)	Not statically checkable	
Dir 2.1 (Req)	Supported	
Dir 3.1 (Req)	Not statically checkable	
Dir 4.1 (Req)	Not statically checkable	
Dir 4.2 (Adv)	Not statically checkable	
Dir 4.3 (Req)	Supported	586
Dir 4.4 (Adv)	Supported	602
Dir 4.5 (Adv)	Supported	9046
Dir 4.6 (Adv)	Partially supported <sup>1</sup>	970
Dir 4.7 (Req)	Supported	534
Dir 4.8 (Adv)	Supported	9045
Dir 4.9 (Adv)	Supported	9026
Dir 4.10 (Req)	Supported	451
Dir 4.11 (Req)	Supported	
Dir 4.12 (Req)	Supported	586
Dir 4.13 (Adv)	Not supported	
Rule 1.1 (Req)	Supported	793
Rule 1.2 (Adv*)	Not statically checkable	
Rule 1.3 (Req*)	Partially supported <sup>2</sup>	2 31 40 54 86 136 401 413 414 415 416 419 424 428 437 449 557 558 564 604 606 629 719 931 2454 9020 9023
Rule 2.1 (Req*)	Supported	527 681 827
Rule 2.2 (Req*)	Supported	438 505 520 521 522 774
Rule 2.3 (Adv)	Supported	751 756
Rule 2.4 (Adv)	Supported	753 9058
Rule 2.5 (Adv)	Supported	750 755
Rule 2.6 (Adv)	Supported	563
Rule 2.7 (Adv)	Supported	715
Rule 3.1 (Req)	Supported	602 9059 9066
Rule 3.2 (Req)	Supported	427
Rule 4.1 (Req)	Supported	9039
Rule 4.2 (Adv)	Supported	584 739 9060

Guideline	Support Level	Diagnostics
Rule 5.1 (Req)	Supported	621
Rule 5.2 (Req)	Supported	621
Rule 5.3 (Req)	Supported	578 621
Rule 5.4 (Req)	Supported	547 621 760
Rule 5.5 (Req)	Partially supported <sup>3</sup>	9095 9096
Rule 5.6 (Req)	Not supported	
Rule 5.7 (Req)	Partially supported <sup>4</sup>	407 631
Rule 5.8 (Req)	Not supported	
Rule 5.9 (Adv)	Not supported	
Rule 6.1 (Req)	Supported	9149
Rule 6.2 (Req)	Supported	9088
Rule 7.1 (Req)	Supported	9001
Rule 7.2 (Req)	Supported	9048
Rule 7.3 (Req)	Supported	620 9057
Rule 7.4 (Req)	Supported	489 1776 1778
Rule 8.1 (Req)	Partially supported <sup>5</sup>	601 808 832
Rule 8.2 (Req)	Supported	18 936 937 955
Rule 8.3 (Req)	Supported	18 9072 9073 9094
Rule 8.4 (Req)	Supported	15 957 9075
Rule 8.5 (Req)	Supported	9004
Rule 8.6 (Req)	Not supported	
Rule 8.7 (Adv)	Supported	765
Rule 8.8 (Req)	Supported	839
Rule 8.9 (Adv)	Supported	9003
Rule 8.10 (Req)	Supported	695 9056
Rule 8.11 (Adv)	Supported	9067
Rule 8.12 (Req)	Supported	488
Rule 8.13 (Adv*)	Supported	818 844 954
Rule 8.14 (Req)	Supported	586
Rule 9.1 (Mand*)	Supported	530 603 644
Rule 9.2 (Req)	Supported	9069
Rule 9.3 (Req)	Supported	9068
Rule 9.4 (Req)	Supported	485
Rule 9.5 (Req)	Supported	9054
Rule 10.1 (Req)	Supported	48 9027
Rule 10.2 (Req)	Supported	9028
Rule 10.3 (Req)	Supported	9034
Rule 10.4 (Req)	Supported	9029
Rule 10.5 (Adv)	Supported	9030
Rule 10.6 (Req)	Supported	9031
Rule 10.7 (Req)	Supported	9032
Rule 10.8 (Req)	Supported	9033
Rule 11.1 (Req)	Supported	176 178 9074
Rule 11.2 (Req)	Supported	9076
Rule 11.3 (Req)	Supported	9087
Rule 11.4 (Adv)	Supported	9078
Rule 11.5 (Adv)	Supported	9079
Rule 11.6 (Req)	Supported	923

Guideline	Support Level	Diagnostics
Rule 11.7 (Req)	Supported	<a href="#">177 179 9295</a>
Rule 11.8 (Req)	Supported	<a href="#">9005</a>
Rule 11.9 (Req)	Supported	<a href="#">9080</a>
Rule 12.1 (Adv)	Supported	<a href="#">9050 9097</a>
Rule 12.2 (Req*)	Supported	<a href="#">9053</a>
Rule 12.3 (Adv)	Supported	<a href="#">9008</a>
Rule 12.4 (Adv)	Not supported	
Rule 13.1 (Req*)	Supported	<a href="#">446</a>
Rule 13.2 (Req*)	Supported	<a href="#">564</a>
Rule 13.3 (Adv)	Supported	<a href="#">9049</a>
Rule 13.4 (Adv)	Supported	<a href="#">720 820 9084</a>
Rule 13.5 (Req*)	Supported	<a href="#">9007</a>
Rule 13.6 (Mand)	Partially supported <sup>6</sup>	<a href="#">9006</a>
Rule 14.1 (Req*)	Supported	<a href="#">9009</a>
Rule 14.2 (Req*)	Not supported	
Rule 14.3 (Req*)	Supported	<a href="#">650 685 774</a>
Rule 14.4 (Req)	Supported	<a href="#">9036</a>
Rule 15.1 (Adv)	Supported	<a href="#">801</a>
Rule 15.2 (Req)	Supported	<a href="#">9064</a>
Rule 15.3 (Req)	Supported	<a href="#">9041</a>
Rule 15.4 (Adv)	Supported	<a href="#">9011</a>
Rule 15.5 (Adv)	Supported	<a href="#">904</a>
Rule 15.6 (Req)	Supported	<a href="#">9012</a>
Rule 15.7 (Req)	Supported	<a href="#">9013 9063</a>
Rule 16.1 (Req)	Supported	<a href="#">616 744 764 825 9014 9042 9077 9081 9082 9085</a>
Rule 16.2 (Req)	Supported	<a href="#">44 9055</a>
Rule 16.3 (Req)	Supported	<a href="#">616 825 9077 9090</a>
Rule 16.4 (Req)	Supported	<a href="#">744 9014 9085</a>
Rule 16.5 (Req)	Supported	<a href="#">9082</a>
Rule 16.6 (Req)	Supported	<a href="#">764 9081</a>
Rule 16.7 (Req)	Supported	<a href="#">483</a>
Rule 17.1 (Req)	Supported	<a href="#">829</a>
Rule 17.2 (Req*)	Supported	<a href="#">9070</a>
Rule 17.3 (Mand)	Supported	<a href="#">718</a>
Rule 17.4 (Mand)	Supported	<a href="#">533</a>
Rule 17.5 (Adv*)	Supported	<a href="#">473</a>
Rule 17.6 (Mand)	Supported	<a href="#">9043</a>
Rule 17.7 (Req)	Supported	<a href="#">534</a>
Rule 17.8 (Adv*)	Supported	<a href="#">9044</a>
Rule 18.1 (Req*)	Supported	<a href="#">415 416 428 661 662 676</a>
Rule 18.2 (Req*)	Assistance provided <sup>7</sup>	<a href="#">947</a>
Rule 18.3 (Req*)	Assistance provided <sup>8</sup>	<a href="#">946</a>
Rule 18.4 (Adv)	Supported	<a href="#">9016</a>
Rule 18.5 (Adv)	Supported	<a href="#">9025</a>
Rule 18.6 (Req*)	Supported	<a href="#">604 733 789</a>
Rule 18.7 (Req)	Supported	<a href="#">9038</a>

Guideline	Support Level	Diagnostics
Rule 18.8 (Req)	Supported	9035
Rule 19.1 (Mand*)	Not supported	
Rule 19.2 (Adv)	Supported	9018
Rule 20.1 (Adv)	Supported	9019
Rule 20.2 (Req)	Supported	9020
Rule 20.3 (Req)	Supported	12 544
Rule 20.4 (Req)	Supported	9051
Rule 20.5 (Adv)	Supported	9021
Rule 20.6 (Req)	Supported	436
Rule 20.7 (Req)	Partially supported <sup>9</sup>	665
Rule 20.8 (Req)	Supported	9037
Rule 20.9 (Req)	Supported	553
Rule 20.10 (Adv)	Supported	9024
Rule 20.11 (Req)	Supported	484
Rule 20.12 (Req)	Supported	9015
Rule 20.13 (Req)	Supported	16 544 9160
Rule 20.14 (Req)	Supported	8
Rule 21.1 (Req)	Supported	136 9071 9083
Rule 21.2 (Req)	Supported	9093
Rule 21.3 (Req)	Supported	586
Rule 21.4 (Req)	Supported	829
Rule 21.5 (Req)	Supported	586 829
Rule 21.6 (Req)	Supported	586
Rule 21.7 (Req)	Supported	586
Rule 21.8 (Req)	Supported	586
Rule 21.9 (Req)	Supported	586
Rule 21.10 (Req)	Supported	586 829
Rule 21.11 (Adv)	Supported	829
Rule 21.12 (Req)	Supported	586 829
Rule 22.1 (Req*)	Supported	429 698
Rule 22.2 (Mand*)	Supported	424 449
Rule 22.3 (Req*)	Not supported	
Rule 22.4 (Mand*)	Supported	2477
Rule 22.5 (Mand*)	Supported	9047
Rule 22.6 (Mand*)	Supported	2471

<sup>1</sup>The exemption for bit-fields is not currently honored<sup>2</sup>PC-lint Plus will warn about many, but not all, undefined/unspecified behaviors<sup>3</sup>Non-identical name clashes between macro names and symbol names are not reported<sup>4</sup>Reports inconsistent tag use and different definitions of tags defined at file scope<sup>5</sup>Function declarations without a parameter list are not reported<sup>6</sup>Potential side effects of evaluating a variable-length array type in a sizeof expression are not diagnosed<sup>7</sup>All pointer subtraction operations are reported<sup>8</sup>All relational operators involving pointers are reported<sup>9</sup>Only expressions that involve binary operators are reported

## 12.2 MISRA C 2012 AMD-1

### 12.2.1 MISRA C 2012 AMD-1 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	6	42.9%
Comprehensive	6	42.9%
Partial	0	0.0%
Assistance	0	0.0%
Not Supported	8	57.1%
Not Statically Checkable	0	
TOTAL	14	

### 12.2.2 MISRA C 2012 AMD-1 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Dir 4.14 (Req)	Not supported	
Rule 12.5 (Mand)	Supported	682 882
Rule 21.13 (Mand*)	Supported	426
Rule 21.14 (Req*)	Not supported	
Rule 21.15 (Req)	Supported	857
Rule 21.16 (Req)	Supported	9098
Rule 21.17 (Mand*)	Supported	419 420
Rule 21.18 (Mand*)	Supported	419 420 422
Rule 21.19 (Mand*)	Not supported	
Rule 21.20 (Mand*)	Not supported	
Rule 22.7 (Req*)	Not supported	
Rule 22.8 (Req*)	Not supported	
Rule 22.9 (Req*)	Not supported	
Rule 22.10 (Req*)	Not supported	

## 12.3 MISRA C 2012 AMD-2

### 12.3.1 MISRA C 2012 AMD-2 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	4	100.0%
Comprehensive	4	100.0%
Partial	0	0.0%
Assistance	0	0.0%
Not Supported	0	0.0%
Not Statically Checkable	0	
TOTAL	4	

### 12.3.2 MISRA C 2012 AMD-2 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Rule 1.4 (Req)	Supported	<a href="#">586</a>
Rule 21.3 (Req)	Supported	<a href="#">586</a>
Rule 21.8 (Req)	Supported	<a href="#">586</a>
Rule 21.21 (Req)	Supported	<a href="#">586</a>

## 12.4 MISRA C 2012 AMD-3

### 12.4.1 MISRA C 2012 AMD-3 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	21	87.5%
Comprehensive	20	83.3%
Partial	1	4.2%
Assistance	0	0.0%
Not Supported	3	12.5%
Not Statically Checkable	0	
TOTAL	24	

### 12.4.2 MISRA C 2012 AMD-3 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Dir 4.5 (Req)	Not supported	
Rule 1.5 (Req*)	Not supported	
Rule 6.3 (Req)	Supported	<a href="#">9202</a>
Rule 7.5 (Mand)	Supported	<a href="#">2505</a>
Rule 8.15 (Req)	Partially supported <sup>1</sup>	<a href="#">2502</a> <a href="#">9503</a>
Rule 8.16 (Adv)	Supported	<a href="#">9203</a>
Rule 8.17 (Adv)	Supported	<a href="#">9205</a>
Rule 17.9 (Mand*)	Supported	<a href="#">2436</a>
Rule 17.10 (Req)	Supported	<a href="#">841</a>
Rule 17.11 (Adv*)	Supported	<a href="#">2707</a>
Rule 17.12 (Adv)	Supported	<a href="#">9147</a>
Rule 17.13 (Req)	Supported	<a href="#">2417</a> <a href="#">4175</a> <a href="#">5805</a>
Rule 18.9 (Req)	Not supported	
Rule 21.22 (Mand)	Supported	<a href="#">9504</a>
Rule 21.23 (Req)	Supported	<a href="#">9218</a>
Rule 21.24 (Req)	Supported	<a href="#">586</a>
Rule 23.1 (Adv)	Supported	<a href="#">9211</a>
Rule 23.2 (Req)	Supported	<a href="#">2419</a> <a href="#">9213</a>
Rule 23.3 (Adv)	Supported	<a href="#">9208</a>
Rule 23.4 (Req)	Supported	<a href="#">2416</a>
Rule 23.5 (Adv)	Supported	<a href="#">9214</a>
Rule 23.6 (Req)	Supported	<a href="#">9216</a>
Rule 23.7 (Adv)	Supported	<a href="#">9219</a>
Rule 23.8 (Req)	Supported	<a href="#">9210</a>

<sup>1</sup>Differences are only reported if the alignment values or types are different



## 12.5 MISRA C 2012 AMD-4

### 12.5.1 MISRA C 2012 AMD-4 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	10	45.5%
Comprehensive	10	45.5%
Partial	0	0.0%
Assistance	0	0.0%
Not Supported	12	54.5%
Not Statically Checkable	0	
TOTAL	22	

### 12.5.2 MISRA C 2012 AMD-4 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Dir 5.1 (Req)	Supported	<a href="#">457</a>
Dir 5.2 (Req)	Not supported	
Dir 5.3 (Req)	Not supported	
Rule 2.8 (Adv)	Supported	<a href="#">528 529 714 754 768</a>
Rule 7.6 (Req)	Not supported	
Rule 9.6 (Req)	Supported	<a href="#">2903 2904</a>
Rule 9.7 (Mand*)	Not supported	
Rule 11.10 (Req)	Supported	<a href="#">4175</a>
Rule 12.6 (Req)	Supported	<a href="#">181</a>
Rule 18.10 (Mand)	Supported	<a href="#">2708</a>
Rule 21.25 (Req)	Not supported	
Rule 21.26 (Req*)	Not supported	
Rule 22.11 (Req*)	Not supported	
Rule 22.12 (Mand*)	Not supported	
Rule 22.13 (Req)	Supported	<a href="#">2751</a>
Rule 22.14 (Mand*)	Not supported	
Rule 22.15 (Req*)	Not supported	
Rule 22.16 (Req*)	Supported	<a href="#">454</a>
Rule 22.17 (Req*)	Supported	<a href="#">455</a>
Rule 22.18 (Req*)	Supported	<a href="#">2531</a>
Rule 22.19 (Req*)	Not supported	
Rule 22.20 (Mand*)	Not supported	

## 12.6 MISRA C++

### 12.6.1 MISRA C++ Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	181	83.8%
Comprehensive	176	81.5%
Partial	4	1.9%
Assistance	1	0.5%
Not Supported	35	16.2%
Not Statically Checkable	12	
TOTAL	228	

### 12.6.2 MISRA C++ Guideline Support Matrix

Guideline	Support Level	Diagnostics
Rule 0-1-1 (req)	Supported	527 681 685 774 827 944
Rule 0-1-2 (req)	Supported	685 774 827 944
Rule 0-1-3 (req)	Supported	528 529 714 752 757 1715
Rule 0-1-4 (req)	Supported	528 529 550 551 552
Rule 0-1-5 (req)	Supported	751 753 756 758
Rule 0-1-6 (req)	Supported	438 838
Rule 0-1-7 (req)	Supported	534
Rule 0-1-8 (req)	Supported	9175
Rule 0-1-9 (req)	Supported	438 587 685 774 838 944 948
Rule 0-1-10 (req)	Supported	528 714 1714 1914
Rule 0-1-11 (req)	Supported	715 9215
Rule 0-1-12 (req)	Supported	715 9215
Rule 0-2-1 (req)	Not supported	
Rule 0-3-1 (doc)	Supported	
Rule 0-3-2 (req)	Supported	534
Rule 0-4-1 (doc)	Not statically checkable	
Rule 0-4-2 (doc)	Not statically checkable	
Rule 0-4-3 (doc)	Not statically checkable	
Rule 1-0-1 (req)	Not supported	
Rule 1-0-2 (req)	Not statically checkable	
Rule 1-0-3 (doc)	Not statically checkable	
Rule 2-2-1 (doc)	Not statically checkable	
Rule 2-3-1 (req)	Supported	584 739
Rule 2-5-1 (adv)	Supported	9102
Rule 2-7-1 (req)	Supported	602
Rule 2-7-2 (req)	Not statically checkable	
Rule 2-7-3 (adv)	Not statically checkable	
Rule 2-10-1 (req)	Not supported	
Rule 2-10-2 (req)	Supported	578
Rule 2-10-3 (req)	Not supported	
Rule 2-10-4 (req)	Not supported	
Rule 2-10-5 (adv)	Supported	9103

Guideline	Support Level	Diagnostics
Rule 2-10-6 (req)	Supported	18
Rule 2-13-1 (req)	Supported	606
Rule 2-13-2 (req)	Supported	9001 9104
Rule 2-13-3 (req)	Supported	9105
Rule 2-13-4 (req)	Supported	9106
Rule 2-13-5 (req)	Supported	707
Rule 3-1-1 (req)	Supported	9107
Rule 3-1-2 (req)	Supported	1798 9108
Rule 3-1-3 (req)	Supported	9067
Rule 3-2-1 (req)	Supported	18 31
Rule 3-2-2 (req)	Supported	15 31
Rule 3-2-3 (req)	Supported	9004
Rule 3-2-4 (req)	Supported	15 31
Rule 3-3-1 (req)	Supported	759 765
Rule 3-3-2 (req)	Supported	401 839
Rule 3-4-1 (req)	Partially supported <sup>1</sup>	9003
Rule 3-9-1 (req)	Partially supported <sup>2</sup>	9073 9094 9168
Rule 3-9-2 (adv)	Supported	970
Rule 3-9-3 (req)	Supported	2498 2499 9110
Rule 4-5-1 (req)	Supported	9111
Rule 4-5-2 (req)	Not supported	
Rule 4-5-3 (req)	Supported	9112
Rule 4-10-1 (req)	Not supported	
Rule 4-10-2 (req)	Supported	910
Rule 5-0-1 (req)	Supported	564
Rule 5-0-2 (adv)	Supported	9113
Rule 5-0-3 (req)	Supported	9114 9116
Rule 5-0-4 (req)	Supported	9117
Rule 5-0-5 (req)	Supported	9115 9118
Rule 5-0-6 (req)	Supported	9119 9120
Rule 5-0-7 (req)	Supported	9121 9122
Rule 5-0-8 (req)	Supported	9123 9124
Rule 5-0-9 (req)	Supported	9125
Rule 5-0-10 (req)	Supported	9126
Rule 5-0-11 (req)	Supported	9128
Rule 5-0-12 (req)	Not supported	
Rule 5-0-13 (req)	Supported	9177
Rule 5-0-14 (req)	Supported	9178
Rule 5-0-15 (req)	Supported	947 9016
Rule 5-0-16 (req)	Supported	415 416 661 662
Rule 5-0-17 (req)	Assistance provided <sup>3</sup>	947
Rule 5-0-18 (req)	Supported	946
Rule 5-0-19 (req)	Supported	9025
Rule 5-0-20 (req)	Supported	9172
Rule 5-0-21 (req)	Supported	9130
Rule 5-2-1 (req)	Supported	9131
Rule 5-2-2 (req)	Supported	1774 1939
Rule 5-2-3 (adv)	Supported	9171

Guideline	Support Level	Diagnostics
Rule 5-2-4 (req)	Supported	1924 1946
Rule 5-2-5 (req)	Supported	9005
Rule 5-2-6 (req)	Supported	611
Rule 5-2-7 (req)	Supported	9176
Rule 5-2-8 (adv)	Supported	9010 9079
Rule 5-2-9 (adv)	Supported	9091
Rule 5-2-10 (adv)	Supported	9049
Rule 5-2-11 (req)	Supported	1753
Rule 5-2-12 (req)	Supported	9132
Rule 5-3-1 (req)	Supported	9133
Rule 5-3-2 (req)	Supported	9134
Rule 5-3-3 (req)	Supported	9135
Rule 5-3-4 (req)	Supported	9006
Rule 5-8-1 (req)	Supported	9136
Rule 5-14-1 (req)	Supported	9007
Rule 5-17-1 (req)	Not supported	
Rule 5-18-1 (req)	Supported	9008
Rule 5-19-1 (adv)	Not supported	
Rule 6-2-1 (req)	Supported	720 820 9084
Rule 6-2-2 (req)	Supported	777 9252
Rule 6-2-3 (req)	Supported	9138
Rule 6-3-1 (req)	Supported	9012
Rule 6-4-1 (req)	Supported	9012
Rule 6-4-2 (req)	Supported	9013
Rule 6-4-3 (req)	Supported	9042
Rule 6-4-4 (req)	Supported	9055
Rule 6-4-5 (req)	Supported	9090
Rule 6-4-6 (req)	Supported	744 787 9139
Rule 6-4-7 (req)	Supported	483
Rule 6-4-8 (req)	Supported	764
Rule 6-5-1 (req)	Not supported	
Rule 6-5-2 (req)	Not supported	
Rule 6-5-3 (req)	Supported	850
Rule 6-5-4 (req)	Not supported	
Rule 6-5-5 (req)	Not supported	
Rule 6-5-6 (req)	Not supported	
Rule 6-6-1 (req)	Supported	9041
Rule 6-6-2 (req)	Supported	107 9064
Rule 6-6-3 (req)	Not supported	
Rule 6-6-4 (req)	Supported	9011
Rule 6-6-5 (req)	Supported	904
Rule 7-1-1 (req)	Supported	843 952 953
Rule 7-1-2 (req)	Supported	818 1764
Rule 7-2-1 (req)	Not supported	
Rule 7-3-1 (req)	Supported	9141 9162
Rule 7-3-2 (req)	Supported	9142
Rule 7-3-3 (req)	Supported	1751
Rule 7-3-4 (req)	Supported	9144

Guideline	Support Level	Diagnostics
Rule 7-3-5 (req)	Not supported	
Rule 7-3-6 (req)	Supported	<a href="#">9145</a>
Rule 7-4-1 (doc)	Not statically checkable	
Rule 7-4-2 (req)	Not supported	
Rule 7-4-3 (req)	Not supported	
Rule 7-5-1 (req)	Supported	<a href="#">604</a>
Rule 7-5-2 (req)	Supported	<a href="#">789</a>
Rule 7-5-3 (req)	Supported	<a href="#">1780</a> <a href="#">1940</a>
Rule 7-5-4 (req)	Supported	<a href="#">9070</a>
Rule 8-0-1 (req)	Supported	<a href="#">9146</a>
Rule 8-3-1 (req)	Supported	<a href="#">1735</a>
Rule 8-4-1 (req)	Supported	<a href="#">9165</a>
Rule 8-4-2 (req)	Supported	<a href="#">9072</a> <a href="#">9272</a>
Rule 8-4-3 (req)	Supported	<a href="#">533</a>
Rule 8-4-4 (req)	Supported	<a href="#">9147</a>
Rule 8-5-1 (req)	Supported	<a href="#">530</a>
Rule 8-5-2 (req)	Supported	<a href="#">940</a>
Rule 8-5-3 (req)	Supported	<a href="#">9148</a>
Rule 9-3-1 (req)	Supported	<a href="#">605</a> <a href="#">1536</a>
Rule 9-3-2 (req)	Supported	<a href="#">1536</a>
Rule 9-3-3 (req)	Supported	<a href="#">1762</a>
Rule 9-5-1 (req)	Supported	<a href="#">9018</a>
Rule 9-6-1 (doc)	Not statically checkable	
Rule 9-6-2 (req)	Supported	<a href="#">9149</a>
Rule 9-6-3 (req)	Supported	<a href="#">9149</a>
Rule 9-6-4 (req)	Supported	<a href="#">9088</a>
Rule 10-1-1 (adv)	Supported	<a href="#">9174</a>
Rule 10-1-2 (req)	Not supported	
Rule 10-1-3 (req)	Supported	<a href="#">1748</a>
Rule 10-2-1 (adv)	Not supported	
Rule 10-3-1 (req)	Not supported	
Rule 10-3-2 (req)	Supported	<a href="#">1909</a>
Rule 10-3-3 (req)	Supported	<a href="#">9170</a>
Rule 11-0-1 (req)	Supported	<a href="#">9150</a>
Rule 12-1-1 (req)	Supported	<a href="#">1506</a>
Rule 12-1-2 (adv)	Supported	<a href="#">1928</a>
Rule 12-1-3 (req)	Supported	<a href="#">9169</a>
Rule 12-8-1 (req)	Partially supported <sup>4</sup>	<a href="#">1938</a>
Rule 12-8-2 (req)	Supported	<a href="#">9151</a>
Rule 14-5-1 (req)	Not supported	
Rule 14-5-2 (req)	Supported	<a href="#">1789</a>
Rule 14-5-3 (req)	Supported	<a href="#">1797</a>
Rule 14-6-1 (req)	Not supported	
Rule 14-6-2 (req)	Not supported	
Rule 14-7-1 (req)	Not supported	
Rule 14-7-2 (req)	Not supported	
Rule 14-7-3 (req)	Partially supported <sup>5</sup>	<a href="#">1576</a>
Rule 14-8-1 (req)	Not supported	

Guideline	Support Level	Diagnostics
Rule 14-8-2 (adv)	Supported	<a href="#">9153</a>
Rule 15-0-1 (doc)	Not statically checkable	
Rule 15-0-2 (adv)	Supported	<a href="#">9154</a>
Rule 15-0-3 (req)	Supported	<a href="#">646</a>
Rule 15-1-1 (req)	Not supported	
Rule 15-1-2 (req)	Supported	<a href="#">1419</a>
Rule 15-1-3 (req)	Supported	<a href="#">9156</a>
Rule 15-3-1 (req)	Supported	<a href="#">1546</a>
Rule 15-3-2 (adv)	Not supported	
Rule 15-3-3 (req)	Not supported	
Rule 15-3-4 (req)	Supported	<a href="#">1560</a>
Rule 15-3-5 (req)	Supported	<a href="#">1752</a>
Rule 15-3-6 (req)	Supported	<a href="#">1775</a>
Rule 15-3-7 (req)	Supported	<a href="#">1127</a>
Rule 15-4-1 (req)	Supported	<a href="#">1548</a>
Rule 15-5-1 (req)	Supported	<a href="#">1546</a>
Rule 15-5-2 (req)	Supported	<a href="#">1549</a>
Rule 15-5-3 (req)	Supported	<a href="#">1546</a>
Rule 16-0-1 (req)	Supported	<a href="#">9019</a>
Rule 16-0-2 (req)	Supported	<a href="#">9158</a> <a href="#">9159</a>
Rule 16-0-3 (req)	Supported	<a href="#">9021</a>
Rule 16-0-4 (req)	Supported	<a href="#">9026</a>
Rule 16-0-5 (req)	Supported	<a href="#">436</a>
Rule 16-0-6 (req)	Supported	<a href="#">9022</a>
Rule 16-0-7 (req)	Supported	<a href="#">553</a>
Rule 16-0-8 (req)	Supported	<a href="#">16</a> <a href="#">544</a> <a href="#">9160</a>
Rule 16-1-1 (req)	Supported	<a href="#">491</a>
Rule 16-1-2 (req)	Supported	<a href="#">8</a>
Rule 16-2-1 (req)	Not supported	
Rule 16-2-2 (req)	Not supported	
Rule 16-2-3 (req)	Supported	<a href="#">967</a>
Rule 16-2-4 (req)	Supported	<a href="#">9020</a>
Rule 16-2-5 (adv)	Supported	<a href="#">9020</a>
Rule 16-2-6 (req)	Supported	<a href="#">12</a>
Rule 16-3-1 (req)	Supported	<a href="#">9023</a>
Rule 16-3-2 (adv)	Supported	<a href="#">9024</a>
Rule 16-6-1 (doc)	Not statically checkable	
Rule 17-0-1 (req)	Supported	<a href="#">9052</a> <a href="#">9071</a>
Rule 17-0-2 (req)	Supported	<a href="#">9093</a>
Rule 17-0-3 (req)	Not supported	
Rule 17-0-4 (req)	Supported	
Rule 17-0-5 (req)	Supported	<a href="#">586</a>
Rule 18-0-1 (req)	Supported	<a href="#">829</a>
Rule 18-0-2 (req)	Supported	<a href="#">586</a>
Rule 18-0-3 (req)	Supported	<a href="#">586</a>
Rule 18-0-4 (req)	Supported	<a href="#">829</a>
Rule 18-0-5 (req)	Supported	<a href="#">586</a>
Rule 18-2-1 (req)	Supported	<a href="#">586</a>

Guideline	Support Level	Diagnostics
Rule 18-4-1 (req)	Supported	<a href="#">586 9173</a>
Rule 18-7-1 (req)	Supported	<a href="#">829</a>
Rule 19-3-1 (req)	Supported	<a href="#">586</a>
Rule 27-0-1 (req)	Supported	<a href="#">829</a>

<sup>1</sup>Only global objects with external linkage that could be defined within a function are reported

<sup>2</sup>PC-lint Plus reports on differences in type alias names, not "token-for-token" differences.

<sup>3</sup>All pointer subtraction operations are reported

<sup>4</sup>Reports on all constructors instead of just copy constructors and reports accesses of global data, not just modifications

<sup>5</sup>Only violations involving function template specializations are reported

## 12.7 MISRA C 2004

### 12.7.1 MISRA C 2004 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	121	92.4%
Comprehensive	121	92.4%
Partial	0	0.0%
Assistance	0	0.0%
Not Supported	10	7.6%
Not Statically Checkable	11	
TOTAL	142	

### 12.7.2 MISRA C 2004 Guideline Support Matrix

Guideline	Support Level	Diagnostics
Rule 1.1 (req)	Supported	
Rule 1.2 (req)	Supported	1 2 31 40 54 86 136 401 413 414 415 416 419 424 428 437 449 557 558 564 604 629 719 931 2454 5125 5537
Rule 1.3 (req)	Not statically checkable	
Rule 1.4 (req)	Not statically checkable	
Rule 1.5 (adv)	Not statically checkable	
Rule 2.1 (req)	Supported	586
Rule 2.2 (req)	Supported	9260
Rule 2.3 (req)	Supported	602
Rule 2.4 (adv)	Supported	602
Rule 3.1 (req)	Not statically checkable	
Rule 3.2 (req)	Not statically checkable	
Rule 3.3 (adv)	Not statically checkable	
Rule 3.4 (req)	Supported	975
Rule 3.5 (req)	Not statically checkable	
Rule 3.6 (req)	Supported	
Rule 4.1 (req)	Supported	606 2006 9104 9204
Rule 4.2 (req)	Supported	584 739
Rule 5.1 (req)	Not supported	
Rule 5.2 (req)	Supported	578
Rule 5.3 (req)	Not supported	
Rule 5.4 (req)	Not supported	
Rule 5.5 (adv)	Not supported	
Rule 5.6 (adv)	Not supported	
Rule 5.7 (adv)	Not supported	
Rule 6.1 (req)	Supported	9128 9209
Rule 6.2 (req)	Supported	9128
Rule 6.3 (adv)	Supported	970
Rule 6.4 (req)	Supported	9212
Rule 6.5 (req)	Supported	9088 9288
Rule 7.1 (req)	Supported	9001 9104



Guideline	Support Level	Diagnostics
Rule 8.1 (req)	Supported	718 746 937 957
Rule 8.2 (req)	Supported	601 808
Rule 8.3 (req)	Supported	9073 9094
Rule 8.4 (req)	Supported	15 18
Rule 8.5 (req)	Supported	9107
Rule 8.6 (req)	Supported	9108
Rule 8.7 (req)	Supported	9003
Rule 8.8 (req)	Supported	9004
Rule 8.9 (req)	Not supported	
Rule 8.10 (req)	Supported	765
Rule 8.11 (req)	Supported	401 839
Rule 8.12 (req)	Supported	9067
Rule 9.1 (req)	Supported	530 644
Rule 9.2 (req)	Supported	576 940 9068
Rule 9.3 (req)	Supported	9148
Rule 10.1 (req)	Supported	9225 9226
Rule 10.2 (req)	Supported	9227 9228
Rule 10.3 (req)	Supported	9229
Rule 10.4 (req)	Supported	9230
Rule 10.5 (req)	Supported	9231
Rule 10.6 (req)	Supported	9048
Rule 11.1 (req)	Supported	176 178 9237
Rule 11.2 (req)	Supported	177 179
Rule 11.3 (adv)	Supported	923
Rule 11.4 (adv)	Supported	9087 9287
Rule 11.5 (req)	Supported	9005
Rule 12.1 (adv)	Supported	9050
Rule 12.2 (req)	Supported	564
Rule 12.3 (req)	Supported	9006
Rule 12.4 (req)	Supported	9007
Rule 12.5 (req)	Supported	9240
Rule 12.6 (adv)	Supported	9232
Rule 12.7 (req)	Supported	9233
Rule 12.8 (req)	Supported	9234
Rule 12.9 (req)	Supported	9235
Rule 12.10 (req)	Supported	9008
Rule 12.11 (adv)	Not supported	
Rule 12.12 (req)	Supported	9110
Rule 12.13 (adv)	Supported	9049
Rule 13.1 (req)	Supported	720 9236
Rule 13.2 (adv)	Supported	9224
Rule 13.3 (req)	Supported	777 9252
Rule 13.4 (req)	Supported	9009
Rule 13.5 (req)	Supported	440 443
Rule 13.6 (req)	Supported	850
Rule 13.7 (req)	Supported	650 685 774 845
Rule 14.1 (req)	Supported	527 681 827
Rule 14.2 (req)	Supported	505 522

Guideline	Support Level	Diagnostics
Rule 14.3 (req)	Supported	<a href="#">9138</a>
Rule 14.4 (req)	Supported	<a href="#">801</a>
Rule 14.5 (req)	Supported	<a href="#">9254</a>
Rule 14.6 (req)	Supported	<a href="#">9011</a>
Rule 14.7 (req)	Supported	<a href="#">904</a>
Rule 14.8 (req)	Supported	<a href="#">9012</a>
Rule 14.9 (req)	Supported	<a href="#">9012</a>
Rule 14.10 (req)	Supported	<a href="#">9013</a> <a href="#">9063</a>
Rule 15.0 (req)	Supported	<a href="#">9042</a>
Rule 15.1 (req)	Supported	<a href="#">44</a> <a href="#">9055</a>
Rule 15.2 (req)	Supported	<a href="#">9090</a>
Rule 15.3 (req)	Supported	<a href="#">9014</a> <a href="#">9139</a>
Rule 15.4 (req)	Supported	<a href="#">9238</a>
Rule 15.5 (req)	Supported	<a href="#">764</a>
Rule 16.1 (req)	Supported	<a href="#">9165</a>
Rule 16.2 (req)	Supported	<a href="#">9070</a>
Rule 16.3 (req)	Supported	<a href="#">955</a>
Rule 16.4 (req)	Supported	<a href="#">9072</a>
Rule 16.5 (req)	Supported	<a href="#">937</a>
Rule 16.6 (req)	Supported	<a href="#">118</a> <a href="#">119</a>
Rule 16.7 (adv)	Supported	<a href="#">818</a>
Rule 16.8 (req)	Supported	<a href="#">533</a>
Rule 16.9 (req)	Supported	<a href="#">9147</a>
Rule 16.10 (req)	Supported	<a href="#">534</a>
Rule 17.1 (req)	Not statically checkable	
Rule 17.2 (req)	Not statically checkable	
Rule 17.3 (req)	Not statically checkable	
Rule 17.4 (req)	Supported	<a href="#">9016</a> <a href="#">9017</a> <a href="#">9264</a>
Rule 17.5 (adv)	Supported	<a href="#">9025</a>
Rule 17.6 (req)	Supported	<a href="#">604</a> <a href="#">733</a> <a href="#">789</a>
Rule 18.1 (req)	Supported	<a href="#">115</a>
Rule 18.2 (req)	Not supported	
Rule 18.3 (req)	Not statically checkable	
Rule 18.4 (req)	Supported	<a href="#">9018</a>
Rule 19.1 (adv)	Supported	<a href="#">9019</a>
Rule 19.2 (adv)	Supported	<a href="#">9020</a>
Rule 19.3 (req)	Supported	<a href="#">12</a>
Rule 19.4 (req)	Not supported	
Rule 19.5 (req)	Supported	<a href="#">9158</a> <a href="#">9159</a>
Rule 19.6 (req)	Supported	<a href="#">9021</a>
Rule 19.7 (adv)	Supported	<a href="#">9026</a>
Rule 19.8 (req)	Supported	<a href="#">131</a>
Rule 19.9 (req)	Supported	<a href="#">436</a>
Rule 19.10 (req)	Supported	<a href="#">9022</a>
Rule 19.11 (req)	Supported	<a href="#">553</a>
Rule 19.12 (req)	Supported	<a href="#">9023</a>
Rule 19.13 (adv)	Supported	<a href="#">9024</a>
Rule 19.14 (req)	Supported	<a href="#">491</a>

Guideline	Support Level	Diagnostics
Rule 19.15 (req)	Supported	451
Rule 19.16 (req)	Supported	16 544 9160
Rule 19.17 (req)	Supported	8
Rule 20.1 (req)	Supported	980 9071 9083
Rule 20.2 (req)	Supported	9093
Rule 20.3 (req)	Supported	
Rule 20.4 (req)	Supported	586
Rule 20.5 (req)	Supported	586
Rule 20.6 (req)	Supported	586
Rule 20.7 (req)	Supported	586
Rule 20.8 (req)	Supported	586 829
Rule 20.9 (req)	Supported	829
Rule 20.10 (req)	Supported	586
Rule 20.11 (req)	Supported	586
Rule 20.12 (req)	Supported	586
Rule 21.1 (req)	Supported	

## 13 AUTOSAR<sup>®</sup> Standard Checking

### 13.1 Introduction to AUTOSAR Support

The AUTOSAR C++ Coding Guidelines are an update to MISRA C++ 2008 intended for use with C++14 in the development of safety-related software. The AUTOSAR guidelines consist of a combination of unmodified MISRA C++ 2008 rules, modified MISRA C++ 2008 rules, and completely new rules. Like its predecessor, the goal of these guidelines is to help facilitate the development of safe and reliable software.

There were five released versions of the AUTOSAR guidelines named after the year and month of their release: 17-03, 17-10, 18-03, 18-10, and 19-03. There are no guideline differences between 18-10 and 19-03. Previous versions of PC-lint Plus targeted AUTOSAR 17-03. Starting with PC-lint Plus 2.0, both AUTOSAR 17-03 and AUTOSAR 19-03 are supported. These are hereafter referred to as AUTOSAR17 and AUTOSAR19, respectively.

PC-lint Plus provides support for detecting violations of the AUTOSAR17 guidelines using the `au-autosar.lnt` file and violations of the AUTOSAR19 guidelines using the `au-autosar19.lnt` file, both distributed with the product in the `lnt/` directory. The tables that follow detail the level of support and the supporting messages for each guideline. While some of the messages are very specific to AUTOSAR guidelines, any of the messages may be employed individually in order to make use of a subset of the checks, outside of AUTOSAR compliance checking.

**The author file enables checks for both library and non-library code. This means that the standard headers employed by your source code are subject to the same scrutiny as the rest of the project. This is often a project requirement but can result in a lot of noise if library code is not subject to the same compliance requirements as the rest of the project. The simplest way to disable AUTOSAR checks for library code is to place the options `-wlib(4) -wlib(1)` immediately after the author file is referenced. This raises and immediately lowers the warning level for libraries resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.**

The following subsections document the level of support provided by PC-lint Plus for each of the AUTOSAR guidelines. The rule number, level of support, and primary enforcing messages are provided. While every effort is made to ensure the correctness of the information provided here, Vector Informatik GmbH makes no guarantee with respect to the accuracy of the information conveyed.

The following terms are used to characterize the support that PC-lint Plus provides each guideline:

- **Supported** - For statically checkable guidelines, the rule is comprehensively supported and no false positives nor false negatives are expected. For guidelines that are not fully statically checkable, substantial support is provided to detect statically checkable violations.
- **Partially Supported** - Meaningful support is provided but there may be cases where false positives and/or false negatives may occur due to limitations in the currently implemented detection method, one or more guideline exceptions are not implemented, etc.
- **Assistance Provided** - While enforcement of the actual guideline is not supported (or cannot be statically checked), *potential* violations of the guideline are diagnosed.
- **Not Supported** - No meaningful level of support is currently provided for this guideline.
- **Not Statically Checkable** - Violations of the guideline cannot be detected by means of static analysis.

Most guidelines marked as *Partially Supported* or *Assistance Provided* contain a footnote that corresponds to an explanation of the corresponding limitation(s) at the end of each support matrix.

---

<sup>®</sup> AUTOSAR is a registered trademark of AUTOSAR GbR.

## 13.2 AUTOSAR19

### 13.2.1 AUTOSAR19 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	216	61.4%
Comprehensive	197	56.0%
Partial	17	4.8%
Assistance	2	0.6%
Not Supported	136	38.6%
Not Statically Checkable	45	
TOTAL	397	

### 13.2.2 AUTOSAR19 Guideline Support Matrix

Guideline	Support Level	Diagnostics
M0-1-1	Supported	527 681 685 774 827 944
M0-1-2	Supported	685 774 827 944
M0-1-3	Supported	528 529 714 752 757 1715
M0-1-4	Supported	528 529 550 551 552
A0-1-1	Partially supported <sup>1</sup>	438 838
A0-1-2	Supported	534
M0-1-8	Supported	9175
M0-1-9	Supported	438 587 685 774 838 944 948
M0-1-10	Supported	528 714 1714 1914
A0-1-3	Partially supported <sup>2</sup>	528
A0-1-4	Supported	715
A0-1-5	Not supported	
A0-1-6	Supported	751 753 756 758
M0-2-1	Not supported	
M0-3-1	Not statically checkable	
M0-3-2	Not statically checkable	
M0-4-1	Not statically checkable	
M0-4-2	Not statically checkable	
A0-4-1	Not statically checkable	
A0-4-2	Supported	586
A0-4-3	Not statically checkable	
A0-4-4	Partially supported <sup>3</sup>	2423
A1-1-1	Partially supported <sup>4</sup>	586 1407 1906
M1-0-2	Not statically checkable	
A1-1-2	Not statically checkable	
A1-1-3	Not statically checkable	
A1-2-1	Not statically checkable	
A1-4-1	Not statically checkable	
A1-4-3	Not statically checkable	
A2-3-1	Not supported	
A2-5-1	Supported	584 739
A2-5-2	Supported	9102

Guideline	Support Level	Diagnostics
M2-7-1	Supported	<a href="#">602</a>
A2-7-1	Supported	<a href="#">427</a>
A2-7-2	Not statically checkable	
A2-7-3	Not supported	
A2-7-5	Not statically checkable	
A2-8-1	Not statically checkable	
A2-8-2	Not statically checkable	
M2-10-1	Not supported	
A2-10-1	Not supported	
A2-10-6	Not supported	
A2-10-4	Not supported	
A2-10-5	Not supported	
A2-11-1	Supported	<a href="#">586</a>
A2-13-1	Supported	<a href="#">606</a>
A2-13-6	Supported	<a href="#">9443</a>
A2-13-5	Supported	<a href="#">9439</a>
M2-13-2	Supported	<a href="#">9001</a> <a href="#">9104</a>
M2-13-3	Supported	<a href="#">9105</a>
M2-13-4	Supported	<a href="#">9106</a>
A2-13-2	Supported	<a href="#">1107</a>
A2-13-3	Supported	<a href="#">586</a>
A2-13-4	Supported	<a href="#">1776</a>
A3-1-1	Supported	<a href="#">9107</a>
A3-1-2	Not supported	
A3-1-3	Not supported	
M3-1-2	Supported	<a href="#">1798</a> <a href="#">9108</a>
A3-1-4	Supported	<a href="#">9067</a>
A3-1-5	Supported	<a href="#">9449</a>
A3-1-6	Not supported	
M3-2-1	Supported	<a href="#">18</a> <a href="#">31</a>
M3-2-2	Supported	<a href="#">15</a> <a href="#">31</a>
M3-2-3	Supported	<a href="#">9004</a>
M3-2-4	Not supported	
A3-3-1	Not supported	
A3-3-2	Not supported	
M3-3-2	Supported	<a href="#">401</a> <a href="#">839</a>
M3-4-1	Partially supported <sup>5</sup>	<a href="#">9003</a>
A3-8-1	Not statically checkable	
M3-9-1	Partially supported	<a href="#">9073</a> <a href="#">9094</a> <a href="#">9168</a>
A3-9-1	Supported	<a href="#">586</a>
M3-9-3	Supported	<a href="#">2498</a> <a href="#">2499</a> <a href="#">9110</a>
M4-5-1	Supported	<a href="#">9111</a>
A4-5-1	Not supported	
M4-5-3	Supported	<a href="#">9112</a>
A4-7-1	Not supported	
M4-10-1	Not supported	
A4-10-1	Supported	<a href="#">910</a>
M4-10-2	Supported	<a href="#">910</a>

Guideline	Support Level	Diagnostics
A5-0-1	Supported	564
M5-0-2	Supported	9113
M5-0-3	Supported	9114 9116
M5-0-4	Supported	9117
M5-0-5	Supported	9115 9118
M5-0-6	Supported	9119 9120
M5-0-7	Supported	9121 9122
M5-0-8	Supported	9123 9124
M5-0-9	Supported	9125
M5-0-10	Supported	9126
M5-0-11	Supported	9128
M5-0-12	Not supported	
A5-0-2	Supported	9177
M5-0-14	Supported	9178
M5-0-15	Supported	947 9016
M5-0-16	Supported	415 416 661 662
M5-0-17	Assistance provided <sup>6</sup>	947
A5-0-4	Not supported	
M5-0-18	Assistance provided <sup>7</sup>	946
A5-0-3	Supported	9025
M5-0-20	Supported	9172
M5-0-21	Supported	9130
A5-1-1	Not supported	
A5-1-2	Not supported	
A5-1-3	Supported	9424
A5-1-4	Not supported	
A5-1-6	Supported	3903
A5-1-7	Supported	9426
A5-1-8	Supported	9442
A5-1-9	Not supported	
M5-2-2	Supported	1774 1939
M5-2-3	Supported	9171
A5-2-1	Supported	586
A5-2-2	Supported	1924 1954
A5-2-3	Supported	9005
M5-2-6	Supported	611
A5-2-4	Supported	586
A5-2-6	Partially supported <sup>8</sup>	9131
M5-2-8	Supported	9010 9079
M5-2-9	Supported	9091
M5-2-10	Supported	9049
M5-2-11	Supported	1753
A5-2-5	Supported	415 416 661 662
M5-2-12	Supported	9132
M5-3-1	Supported	9133
M5-3-2	Supported	9134
M5-3-3	Supported	9135
M5-3-4	Supported	9006



Guideline	Support Level	Diagnostics
A5-3-1	Supported	9414
A5-3-2	Supported	413 613
A5-3-3	Supported	1404
A5-5-1	Not supported	
A5-6-1	Partially supported <sup>9</sup>	414
M5-8-1	Supported	9136
A5-10-1	Not supported	
M5-14-1	Supported	9007
A5-16-1	Not supported	
M5-17-1	Not statically checkable	
M5-18-1	Supported	9008
M5-19-1	Not supported	
M6-2-1	Supported	720 820 9084
A6-2-1	Not supported	
A6-2-2	Not supported	
M6-2-2	Supported	777 9252
M6-2-3	Supported	9138
M6-3-1	Supported	9012
M6-4-1	Supported	9012
M6-4-2	Supported	9013
M6-4-3	Supported	9042
M6-4-4	Supported	9055
M6-4-5	Supported	9090
M6-4-6	Supported	744 787 9139
M6-4-7	Supported	483
A6-4-1	Supported	9181
A6-5-1	Not supported	
A6-5-2	Not supported	
M6-5-2	Not supported	
M6-5-3	Supported	850
M6-5-4	Not supported	
M6-5-5	Not supported	
M6-5-6	Not supported	
A6-5-3	Supported	9456
A6-5-4	Not supported	
A6-6-1	Supported	801
M6-6-1	Supported	9041
M6-6-2	Supported	107 9064
M6-6-3	Not supported	
A7-1-1	Supported	843 952 953
A7-1-2	Not supported	
M7-1-2	Supported	818 1764
A7-1-3	Supported	9183
A7-1-4	Supported	586
A7-1-5	Not supported	
A7-1-6	Supported	9416
A7-1-7	Partially supported <sup>10</sup>	9146
A7-1-8	Supported	618 2618

Guideline	Support Level	Diagnostics
A7-1-9	Supported	9428
A7-2-1	Not supported	
A7-2-2	Supported	9418
A7-2-3	Supported	9419
A7-2-4	Supported	9148
A7-2-5	Not statically checkable	
M7-3-1	Supported	9141 9162
M7-3-2	Supported	9142
M7-3-3	Supported	1751
M7-3-4	Supported	9144
A7-3-1	Not supported	
M7-3-6	Supported	9145
A7-4-1	Supported	586
M7-4-1	Not statically checkable	
M7-4-2	Supported	
M7-4-3	Not supported	
M7-5-1	Supported	604
M7-5-2	Supported	789
A7-5-1	Not supported	
A7-5-2	Supported	9070
A7-6-1	Supported	2436
M8-0-1	Supported	9146
A8-2-1	Not supported	
M8-3-1	Supported	1735
A8-4-1	Supported	9165
M8-4-2	Supported	9072 9272
A8-4-2	Supported	533
M8-4-4	Supported	9147
A8-4-3	Not statically checkable	
A8-4-4	Not supported	
A8-4-5	Not supported	
A8-4-6	Not supported	
A8-4-7	Not supported	
A8-4-8	Not supported	
A8-4-9	Not supported	
A8-4-10	Not supported	
A8-4-11	Not supported	
A8-4-12	Not supported	
A8-4-13	Not supported	
A8-4-14	Not statically checkable	
A8-5-0	Supported	530
A8-5-1	Supported	1729
M8-5-2	Supported	940
A8-5-2	Not supported	
A8-5-3	Supported	9415
A8-5-4	Not supported	
M9-3-1	Not supported	
A9-3-1	Not supported	

Guideline	Support Level	Diagnostics
M9-3-3	Partially supported <sup>11</sup>	1762
A9-5-1	Supported	9018
M9-6-1	Not statically checkable	
A9-6-1	Not supported	
A9-6-2	Not statically checkable	
M9-6-4	Supported	9088
A10-0-1	Not statically checkable	
A10-0-2	Not statically checkable	
A10-1-1	Supported	9432
M10-1-1	Supported	9174
M10-1-2	Not supported	
M10-1-3	Supported	1748
M10-2-1	Not supported	
A10-2-1	Supported	1511
A10-3-1	Supported	9422
A10-3-2	Supported	9421
A10-3-3	Supported	1779
A10-3-5	Supported	9407 9410 9438
M10-3-3	Supported	9170
A10-4-1	Not statically checkable	
M11-0-1	Supported	9150
A11-0-1	Supported	9437
A11-0-2	Not supported	
A11-3-1	Supported	9435
A12-0-1	Not supported	
A12-0-2	Partially supported <sup>12</sup>	1415
A12-1-1	Supported	1401 1928
M12-1-1	Partially supported <sup>13</sup>	1506
A12-1-2	Not supported	
A12-1-3	Not supported	
A12-1-4	Supported	9169
A12-1-5	Not supported	
A12-1-6	Not supported	
A12-4-1	Not supported	
A12-4-2	Supported	9441
A12-6-1	Not supported	
A12-7-1	Not supported	
A12-8-1	Not supported	
A12-8-2	Not supported	
A12-8-3	Not supported	
A12-8-4	Not supported	
A12-8-5	Supported	1529
A12-8-6	Not supported	
A12-8-7	Supported	9185
A13-1-2	Not supported	
A13-1-3	Not supported	
A13-2-1	Supported	9447 9448
A13-2-2	Not supported	

Guideline	Support Level	Diagnostics
A13-2-3	Supported	9186
A13-3-1	Not supported	
A13-5-1	Supported	9187
A13-5-2	Supported	1930
A13-5-3	Supported	1912
A13-5-4	Not supported	
A13-5-5	Supported	9444 9445 9446
A13-6-1	Supported	9440
A14-1-1	Not statically checkable	
A14-5-1	Not supported	
A14-5-2	Not supported	
A14-5-3	Not supported	
M14-5-3	Supported	1797
M14-6-1	Not supported	
A14-7-1	Not supported	
A14-7-2	Partially supported <sup>14</sup>	1576
A14-8-2	Not supported	
A15-0-1	Not statically checkable	
A15-0-2	Not supported	
A15-0-3	Not statically checkable	
A15-0-4	Not statically checkable	
A15-0-5	Not statically checkable	
A15-0-6	Not statically checkable	
A15-0-7	Not supported	
A15-0-8	Not statically checkable	
A15-1-1	Supported	3902
A15-1-2	Supported	9154
M15-0-3	Not supported	
M15-1-1	Not supported	
M15-1-2	Supported	1419
M15-1-3	Supported	9156
A15-1-3	Not supported	
A15-1-4	Not supported	
A15-1-5	Not statically checkable	
A15-2-1	Not supported	
A15-2-2	Not supported	
M15-3-1	Not supported	
A15-3-2	Not statically checkable	
A15-3-3	Not supported	
A15-3-4	Not statically checkable	
M15-3-3	Not supported	
M15-3-4	Not supported	
A15-3-5	Partially supported <sup>15</sup>	1752
M15-3-6	Supported	1775
M15-3-7	Supported	1127
A15-4-1	Supported	1906
A15-4-2	Not supported	
A15-4-3	Not supported	

Guideline	Support Level	Diagnostics
A15-4-4	Not supported	
A15-4-5	Not supported	
A15-5-1	Not supported	
A15-5-2	Supported	586 9156
A15-5-3	Partially supported <sup>16</sup>	9156
A16-0-1	Partially supported <sup>17</sup>	586 886 9021 9026
M16-0-1	Supported	9019
M16-0-2	Supported	9158 9159
M16-0-5	Supported	436
M16-0-6	Supported	9022
M16-0-7	Supported	553
M16-0-8	Supported	16 544 9160
M16-1-1	Supported	491
M16-1-2	Supported	8
M16-2-3	Supported	967
A16-2-1	Supported	9020
A16-2-2	Not supported	
A16-2-3	Not statically checkable	
M16-3-1	Supported	9023
M16-3-2	Supported	9024
A16-6-1	Supported	586 886
A16-7-1	Supported	586 886
A17-0-1	Supported	9093
M17-0-2	Supported	9093
M17-0-3	Not supported	
A17-0-2	Not statically checkable	
M17-0-5	Supported	586
A17-1-1	Not statically checkable	
A17-6-1	Not supported	
A18-0-1	Supported	829
A18-0-2	Partially supported <sup>18</sup>	586
M18-0-3	Supported	586
M18-0-4	Supported	586
M18-0-5	Supported	586
A18-0-3	Supported	586 829
A18-1-1	Supported	9436
A18-1-2	Not supported	
A18-1-3	Not supported	
A18-1-4	Not supported	
A18-1-6	Not supported	
M18-2-1	Supported	586
A18-5-1	Supported	586
A18-5-2	Not supported	
A18-5-3	Supported	424
A18-5-4	Not supported	
A18-5-5	Not supported	
A18-5-6	Not statically checkable	
A18-5-7	Not statically checkable	

Guideline	Support Level	Diagnostics
A18-5-8	Not supported	
A18-5-9	Not supported	
A18-5-10	Not supported	
A18-5-11	Not supported	
M18-7-1	Supported	586
A18-9-1	Not supported	
A18-9-2	Not supported	
A18-9-3	Not supported	
A18-9-4	Not supported	
M19-3-1	Supported	586
A20-8-1	Not supported	
A20-8-2	Not supported	
A20-8-3	Not supported	
A20-8-4	Not supported	
A20-8-5	Not supported	
A20-8-6	Not supported	
A20-8-7	Not statically checkable	
A21-8-1	Not supported	
A23-0-1	Not supported	
A23-0-2	Not supported	
A25-1-1	Not supported	
A25-4-1	Not statically checkable	
A26-5-1	Supported	586
A26-5-2	Not supported	
M27-0-1	Supported	586
A27-0-1	Not statically checkable	
A27-0-4	Not supported	
A27-0-2	Not supported	
A27-0-3	Not supported	

<sup>1</sup>The exemption for loop control variables is not honored

<sup>2</sup>Unused private methods and functions defined in anonymous namespaces are not reported

<sup>3</sup>Many Standard C math functions are checked for arguments that will result in domain errors

<sup>4</sup>The following uses of deprecated features are reported: incrementing an expression of Boolean type, use of the register keyword, use of an exception specification, use of `std::random_shuffle`, calls to `std::ptr_fun`; `std::mem_fun`; or `std::mem_fun_ref`, and calls to `std::bind1st` or `std::bind2nd`. Additionally, many instances of non-conformant syntactic or semantic compilation errors are diagnosed by messages in the ‘error’ message category without appended text citing this rule.

<sup>5</sup>Only global objects with external linkage that could be defined within a function are reported

<sup>6</sup>Reports all pointer subtractions

<sup>7</sup>Reports all pointer relational operations

<sup>8</sup>No allowance is made for unary operators which may result in false positives

<sup>9</sup>Diagnoses cases where PC-lint Plus determines a likely potential for modulus or division by zero

<sup>10</sup>Reports all declarations containing multiple declarators

<sup>11</sup>"could be static" not supported

<sup>12</sup>Calls to functions like `memcpy`, `memcmp`, `memmove` with pointer to non-POD types are reported.

<sup>13</sup>Doesn't report on use of `typeid` or `dynamic_cast` of dynamic type

<sup>14</sup>Function template specializations declared in different files from the primary template are reported

<sup>15</sup>Issued when throwing non-class objects (e.g. pointers)

<sup>16</sup>Doesn't report all cases `std::terminate` is implicitly called

<sup>17</sup>Reports on the use of `#line`, `#error`, `#pragma`, `#undef`, and null directives and function-like macro definitions

<sup>18</sup>The use of `atof`, `atoi`, and `atol` are diagnosed.

## 13.3 AUTOSAR17

### 13.3.1 AUTOSAR17 Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	205	67.0%
Comprehensive	189	61.8%
Partial	14	4.6%
Assistance	2	0.7%
Not Supported	101	33.0%
Not Statically Checkable	30	
TOTAL	336	

### 13.3.2 AUTOSAR17 Guideline Support Matrix

Guideline	Support Level	Diagnostics
M0-1-1	Supported	527 681 685 774 827 944
M0-1-2	Supported	685 774 827 944
M0-1-3	Supported	528 529 714 752 757 1715
M0-1-4	Supported	528 529 550 551 552
M0-1-5	Supported	751 753 756 758
A0-1-1	Partially supported <sup>1</sup>	438 838
A0-1-2	Supported	534
M0-1-8	Supported	9175
M0-1-9	Supported	438 587 685 774 838 944 948
M0-1-10	Supported	528 714 1714 1914
A0-1-3	Partially supported <sup>2</sup>	528
M0-1-11	Supported	715 9215
M0-1-12	Not supported	
M0-2-1	Not supported	
M0-3-1	Not statically checkable	
M0-3-2	Not supported	
M0-4-1	Not statically checkable	
M0-4-2	Not statically checkable	
A0-4-1	Not statically checkable	
A0-4-2	Supported	586
A0-4-3	Not statically checkable	
A1-1-1	Partially supported <sup>3</sup>	586 1407 1906
M1-0-2	Not statically checkable	
A1-1-2	Not statically checkable	
A1-1-3	Not statically checkable	
A1-2-1	Not statically checkable	
A1-4-1	Not supported	
A1-4-2	Not supported	
A2-2-1	Not supported	
A2-5-1	Supported	584 739
A2-6-1	Supported	9102
A2-8-1	Supported	427

Guideline	Support Level	Diagnostics
A2-8-2	Not statically checkable	
A2-8-3	Not supported	
A2-8-4	Supported	1904
A2-9-1	Not supported	
M2-10-1	Not supported	
A2-11-1	Not supported	
M2-10-3	Not supported	
A2-11-2	Not supported	
A2-11-3	Not supported	
A2-11-4	Not supported	
A2-11-5	Not supported	
M2-10-6	Supported	18
A2-14-1	Supported	606
M2-13-2	Supported	9001 9104
M2-13-3	Supported	9105
M2-13-4	Supported	9106
A2-14-2	Supported	1107
A2-14-3	Supported	586
A3-1-1	Supported	9107
A3-1-2	Not supported	
A3-1-3	Not supported	
M3-1-2	Supported	1798 9108
A3-1-4	Supported	9067
M3-2-1	Supported	18 31
M3-2-2	Supported	15 31
M3-2-3	Supported	9004
M3-2-4	Not supported	
A3-3-1	Not supported	
A3-3-2	Supported	1756
M3-3-2	Supported	401 839
M3-4-1	Partially supported <sup>4</sup>	9003
M3-9-1	Partially supported	9073 9094 9168
A3-9-1	Supported	586
M3-9-3	Supported	2498 2499 9110
M4-5-1	Supported	9111
A4-5-1	Not supported	
M4-5-3	Supported	9112
A4-7-1	Not supported	
M4-10-1	Not supported	
A4-10-1	Supported	910
M4-10-2	Supported	910
A5-0-1	Supported	564
M5-0-2	Supported	9113
M5-0-3	Supported	9114 9116
M5-0-4	Supported	9117
M5-0-5	Supported	9115 9118
M5-0-6	Supported	9119 9120
M5-0-7	Supported	9121 9122



Guideline	Support Level	Diagnostics
M5-0-8	Supported	9123 9124
M5-0-9	Supported	9125
M5-0-10	Supported	9126
M5-0-11	Supported	9128
M5-0-12	Not supported	
A5-0-2	Supported	9177
M5-0-14	Supported	9178
M5-0-15	Supported	947 9016
M5-0-16	Supported	415 416 661 662
M5-0-17	Assistance provided <sup>5</sup>	947
M5-0-18	Assistance provided <sup>6</sup>	946
A5-0-3	Supported	9025
M5-0-20	Supported	9172
M5-0-21	Supported	9130
A5-1-1	Not supported	
A5-1-2	Not supported	
A5-1-3	Supported	9424
A5-1-4	Not supported	
A5-1-5	Not supported	
A5-1-6	Supported	3903
A5-1-7	Supported	9426
A5-1-8	Supported	9442
M5-2-1	Supported	9131
M5-2-2	Supported	1774 1939
M5-2-3	Supported	9171
A5-2-1	Supported	586
A5-2-2	Supported	1924 1954
A5-2-3	Supported	9005
M5-2-6	Supported	611
A5-2-4	Supported	586
M5-2-8	Supported	9010 9079
M5-2-9	Supported	9091
M5-2-10	Supported	9049
M5-2-11	Supported	1753
A5-2-5	Supported	415 416 661 662
M5-2-12	Supported	9132
M5-3-1	Supported	9133
M5-3-2	Supported	9134
M5-3-3	Supported	9135
M5-3-4	Supported	9006
A5-3-1	Supported	9414
A5-5-1	Partially supported <sup>7</sup>	414
M5-8-1	Supported	9136
A5-10-1	Not supported	
M5-14-1	Supported	9007
A5-16-1	Not supported	
M5-17-1	Not supported	
M5-18-1	Supported	9008

Guideline	Support Level	Diagnostics
M5-19-1	Not supported	
M6-2-1	Supported	<a href="#">720 820 9084</a>
M6-2-2	Supported	<a href="#">777 9252</a>
M6-2-3	Supported	<a href="#">9138</a>
M6-3-1	Supported	<a href="#">9012</a>
M6-4-1	Supported	<a href="#">9012</a>
M6-4-2	Supported	<a href="#">9013</a>
M6-4-3	Supported	<a href="#">9042</a>
M6-4-4	Supported	<a href="#">9055</a>
M6-4-5	Supported	<a href="#">9090</a>
M6-4-6	Supported	<a href="#">744 787 9139</a>
M6-4-7	Supported	<a href="#">483</a>
A6-4-1	Supported	<a href="#">9181</a>
A6-5-1	Not supported	
A6-5-2	Not supported	
M6-5-2	Not supported	
M6-5-3	Supported	<a href="#">850</a>
M6-5-4	Not supported	
M6-5-5	Not supported	
M6-5-6	Not supported	
A6-6-1	Supported	<a href="#">801</a>
M6-6-1	Supported	<a href="#">9041</a>
M6-6-2	Supported	<a href="#">107 9064</a>
M6-6-3	Not supported	
A7-1-1	Supported	<a href="#">843 952 953</a>
A7-1-2	Not supported	
M7-1-2	Supported	<a href="#">818 1764</a>
A7-1-3	Supported	<a href="#">9183</a>
A7-1-4	Supported	<a href="#">586</a>
A7-1-5	Not supported	
A7-1-6	Supported	<a href="#">9416</a>
A7-1-7	Partially supported <sup>8</sup>	<a href="#">9146</a>
A7-1-8	Supported	<a href="#">618 2618</a>
A7-2-1	Not supported	
A7-2-2	Supported	<a href="#">9418</a>
A7-2-3	Supported	<a href="#">9419</a>
A7-2-4	Supported	<a href="#">9148</a>
M7-3-1	Supported	<a href="#">9141 9162</a>
M7-3-2	Supported	<a href="#">9142</a>
M7-3-3	Supported	<a href="#">1751</a>
M7-3-4	Supported	<a href="#">9144</a>
M7-3-5	Not supported	
M7-3-6	Supported	<a href="#">9145</a>
A7-4-1	Supported	<a href="#">586</a>
M7-4-1	Not statically checkable	
M7-4-2	Supported	
M7-4-3	Not supported	
M7-5-1	Supported	<a href="#">604</a>

Guideline	Support Level	Diagnostics
M7-5-2	Supported	789
A7-5-1	Not supported	
A7-5-2	Supported	9070
M8-0-1	Supported	9146
A8-2-1	Not supported	
M8-3-1	Supported	1735
A8-4-1	Supported	9165
M8-4-2	Supported	9072 9272
A8-4-2	Supported	533
M8-4-4	Supported	9147
M8-5-1	Supported	530
A8-5-1	Supported	1729
M8-5-2	Supported	940
A8-5-2	Not supported	
A8-5-3	Supported	9415
A8-5-4	Not statically checkable	
M9-3-1	Not supported	
A9-3-1	Not supported	
M9-3-3	Partially supported <sup>9</sup>	1762
M9-5-1	Supported	9018
M9-6-1	Not statically checkable	
A9-6-1	Supported	9420
A10-1-1	Supported	9432
M10-1-1	Supported	9174
M10-1-2	Not supported	
M10-1-3	Supported	1748
M10-2-1	Not supported	
A10-2-1	Supported	1511
A10-3-1	Supported	9422
A10-3-2	Supported	9421
A10-3-3	Supported	1779
A10-3-5	Supported	9407 9410 9438
M10-3-3	Supported	9170
M11-0-1	Supported	9150
A11-0-1	Supported	9437
A11-0-2	Not supported	
A11-3-1	Supported	9435
A12-0-1	Not supported	
A12-1-1	Supported	1401 1928
M12-1-1	Partially supported <sup>10</sup>	1506
A12-1-2	Not supported	
A12-1-3	Not supported	
A12-1-4	Supported	9169
A12-4-1	Not supported	
A12-4-2	Supported	9441
A12-6-1	Not supported	
A12-7-1	Not supported	
A12-8-1	Not supported	

Guideline	Support Level	Diagnostics
A12-8-2	Not supported	
A12-8-3	Not supported	
A12-8-4	Not supported	
A12-8-5	Supported	1529
A12-8-6	Not supported	
A12-8-7	Supported	9185
A13-1-1	Supported	9433 9434
A13-1-2	Not supported	
A13-1-3	Not supported	
A13-2-1	Supported	9447 9448
A13-2-2	Not supported	
A13-2-3	Supported	9186
A13-3-1	Not supported	
A13-5-1	Supported	9187
A13-6-1	Supported	9440
A14-1-1	Not statically checkable	
M14-5-2	Supported	1789
M14-5-3	Partially supported	1797
M14-6-1	Not supported	
A14-7-1	Not supported	
M14-7-3	Partially supported <sup>11</sup>	1576
M14-8-1	Not supported	
A14-8-1	Supported	9153
A15-0-1	Not statically checkable	
A15-0-2	Not supported	
A15-0-3	Not statically checkable	
A15-0-4	Not statically checkable	
A15-0-5	Not statically checkable	
A15-0-6	Not statically checkable	
A15-0-7	Not supported	
A15-0-8	Not statically checkable	
A15-1-1	Supported	3902
A15-1-2	Supported	9154
M15-0-3	Not supported	
M15-1-1	Not supported	
M15-1-2	Supported	1419
M15-1-3	Supported	9156
A15-1-3	Not supported	
A15-1-4	Not supported	
A15-1-5	Not statically checkable	
A15-2-1	Not supported	
A15-2-2	Not supported	
M15-3-1	Not supported	
A15-3-1	Not supported	
A15-3-2	Not statically checkable	
A15-3-3	Not supported	
A15-3-4	Not statically checkable	
M15-3-3	Not statically checkable	

Guideline	Support Level	Diagnostics
M15-3-4	Not supported	
A15-3-5	Partially supported <sup>12</sup>	1752
M15-3-6	Supported	1775
M15-3-7	Supported	1127
A15-4-1	Supported	1906
A15-4-2	Not supported	
A15-4-3	Not supported	
A15-4-4	Not supported	
A15-4-5	Not supported	
A15-4-6	Not supported	
A15-5-1	Not supported	
A15-5-2	Supported	586 9156
A15-5-3	Partially supported <sup>13</sup>	9156
A16-0-1	Partially supported <sup>14</sup>	586 886 9021 9026
M16-0-1	Supported	9019
M16-0-2	Supported	9158 9159
M16-0-5	Supported	436
M16-0-6	Supported	9022
M16-0-7	Supported	553
M16-0-8	Supported	16 544 9160
M16-1-1	Supported	491
M16-1-2	Supported	8
M16-2-3	Supported	967
A16-2-1	Supported	9020
A16-2-2	Not supported	
A16-2-3	Not statically checkable	
M16-3-1	Supported	9023
M16-3-2	Supported	9024
A16-6-1	Supported	586 886
A16-7-1	Supported	586 886
A17-0-1	Supported	9093
M17-0-2	Supported	9093
M17-0-3	Not supported	
M17-0-5	Supported	586
A17-1-1	Not statically checkable	
A18-0-1	Supported	829
A18-0-2	Supported	586
M18-0-3	Supported	586
M18-0-4	Supported	586
M18-0-5	Supported	586
A18-0-3	Supported	586 829
A18-1-1	Supported	9436
A18-1-2	Not supported	
A18-1-3	Not supported	
A18-1-4	Not supported	
A18-1-5	Not supported	
M18-2-1	Supported	586
A18-5-1	Supported	586

Guideline	Support Level	Diagnostics
A18-5-2	Not supported	
A18-5-3	Supported	<a href="#">424</a>
A18-5-4	Not supported	
A18-5-5	Not statically checkable	
A18-5-6	Not statically checkable	
A18-5-7	Not statically checkable	
M18-7-1	Supported	<a href="#">586</a>
A18-9-1	Not supported	
A18-9-2	Not supported	
A18-9-3	Not supported	
A18-9-4	Not supported	
M19-3-1	Supported	<a href="#">586</a>
A23-0-1	Not supported	
M27-0-1	Supported	<a href="#">586</a>
A27-0-1	Not statically checkable	
A27-0-2	Not supported	

<sup>1</sup>The exemption for loop control variables is not honored

<sup>2</sup>Unused private methods are not reported

<sup>3</sup>The following uses of deprecated features are reported: incrementing an expression of Boolean type, use of the register keyword, use of an exception specification, use of `std::random_shuffle`, calls to `std::ptr_fun`; `std::mem_fun`; or `std::mem_fun_ref`, and calls to `std::bind1st` or `std::bind2nd`. Additionally, many instances of non-conformant syntactic or semantic compilation errors are diagnosed by messages in the 'error' message category without appended text citing this rule.

<sup>4</sup>Only global objects with external linkage that could be defined within a function are reported

<sup>5</sup>Reports all pointer subtractions

<sup>6</sup>Reports all pointer relational operations

<sup>7</sup>Diagnoses cases where PC-lint Plus determines a likely potential for modulus or division by zero

<sup>8</sup>Reports all declarations containing multiple declarators

<sup>9</sup>"could be static" not supported

<sup>10</sup>Doesn't report on use of `typeid` or `dynamic_cast` of dynamic type

<sup>11</sup>Only violations involving function template specializations are reported

<sup>12</sup>Issued when throwing non-class objects (e.g. pointers)

<sup>13</sup>Doesn't report all cases `std::terminate` is implicitly called

<sup>14</sup>Reports on the use of `#line`, `#error`, `#pragma`, `#undef`, and null directives and function-like macro definitions

## 14 CERT<sup>®</sup> C Standard Checking

### 14.1 Introduction to CERT C Support

The CERT C guidelines consist of “recommendations” and “rules” organized into 17 sections. The name of a guideline consists of a 3-character mnemonic corresponding to the section, a 2-digit number in the range of 00-99, and a suffix (which is always `-C` for CERT C). Guidelines with a number in the range 00-29 are *recommendations* and guidelines in the range 30-99 are *rules*.

PC-lint Plus provides support for detecting violations of the CERT C guidelines using the `au-certc.lnt` file distributed with the product in the `lnt/` directory. The tables that follow detail the level of support and the supporting messages for each guideline. While some of the messages are specific to CERT C guidelines, any of the messages may be employed individually in order to make use of a subset of the checks, outside of CERT C compliance checking.

**The author file enables checks for both library and non-library code. This means that the standard headers employed by your source code are subject to the same scrutiny as the rest of the project. This is often a project requirement but can result in a lot of noise if library code is not subject to the same compliance requirements as the rest of the project. The simplest way to disable CERT C checks for library code is to place the options `-wlib(4) -wlib(1)` immediately after the author file is referenced. This raises and immediately lowers the warning level for libraries resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.**

The theoretical ability for static analysis to detect violations of guidelines vary. Some guidelines (macro replacement lists should be parenthesized) can theoretically always be detected with static analysis methods whereas other guidelines (prefer inline or static functions to function-like macros) are subjective and/or not statically checkable. A guideline is considered to be “Decidable” if all violations of the guidelines can theoretically be detected with static analysis means and not decidable if the rule is either inherently not statically checkable or cannot be diagnosed in all cases.

The following terms are used to characterize the support that PC-lint Plus provides each guideline:

- **Supported** - For statically checkable guidelines, the rule is comprehensively supported and no false positives nor false negatives are expected. For guidelines that are not fully statically checkable, substantial support is provided to detect statically checkable violations.
- **Partially Supported** - Meaningful support is provided but there may be cases where false positives and/or false negatives may occur due to limitations in the currently implemented detection method, one or more guideline exceptions are not implemented, etc.
- **Assistance Provided** - While enforcement of the actual guideline is not supported (or cannot be statically checked), *potential* violations of the guideline are diagnosed.
- **Not Supported** - No meaningful level of support is currently provided for this guideline.
- **Not Statically Checkable** - Violations of the guideline cannot be detected by means of static analysis.

Most guidelines marked as *Partially Supported* or *Assistance Provided* contain a footnote that corresponds to an explanation of the corresponding limitation(s) at the end of this section.

An example of a guideline that is not supported is PRE08-C (Guarantee that header file names are unique). The guideline is statically checkable but PC-lint Plus does not currently have a diagnostic to support the guideline.

An example of “Assistance provided” is PRE00-C (Prefer inline or static functions to function-like macros). The rule is not statically checkable in a meaningful way but PC-lint Plus can diagnose the definition

of function-like macros and the message can be suppressed for those macros deemed to be legitimately necessary.

An example of “Partially supported” is DCL04-C (Do not declare more than one variable per declaration). This guideline is supported except for two exceptions to the recommendation that are not honored by the enforcement mechanism which may result in false positives.

An example of a “Supported” guideline is PRE02-C (Macro replacement lists should be parenthesized). This guideline is decidable and PC-lint Plus fully supports this guideline. Another example is EXP33-C (Do not read uninitialized memory). While this rule is undecidable (and therefore exhaustive support is impossible), PC-lint Plus provides substantial support for the rule and will catch a wide variety of statically checkable violations.

CERT C guideline titles are excerpted from [SEI CERT C Coding Standard](#) © Carnegie Mellon University 1995–2016. See chapter [25](#) for full details.

---

<sup>®</sup>CERT is a registered trademark of Carnegie Mellon University.



## 14.2 CERT C Support

### 14.2.1 CERT C Guideline Support Summary

Category	# Rules	% of Statically Checkable Rules
Supported	152	51.0%
Comprehensive	76	25.5%
Partial	55	18.5%
Assistance	21	7.0%
Not Supported	146	49.0%
Not Statically Checkable	8	
TOTAL	306	

### 14.2.2 CERT C Guideline Support Matrix

Guideline	Description	Support Level	Diagnostics
PRE00-C	Prefer inline or static functions to function-like macros	Assistance provided	<a href="#">9026</a>
PRE01-C	Use parentheses within macros around parameter names	Supported	<a href="#">9022</a>
PRE02-C	Macro replacement lists should be parenthesized	Supported	<a href="#">773</a> <a href="#">973</a>
PRE03-C	Prefer typedefs to defines for encoding non-pointer types	Not supported	
PRE04-C	Do not reuse a standard header file name	Not supported	
PRE05-C	Understand macro replacement when concatenating tokens or performing stringification	Assistance provided <sup>1</sup>	<a href="#">9024</a>
PRE06-C	Enclose header files in an include guard	Supported	<a href="#">967</a>
PRE07-C	Avoid using repeated question marks	Supported	<a href="#">584</a> <a href="#">854</a> <a href="#">9060</a>
PRE08-C	Guarantee that header file names are unique	Not supported	
PRE09-C	Do not replace secure functions with deprecated or obsolescent functions	Not supported	
PRE10-C	Wrap multistatement macros in a do-while loop	Supported	<a href="#">9502</a>
PRE11-C	Do not conclude macro definitions with a semicolon	Supported	<a href="#">823</a>
PRE12-C	Do not define unsafe macros	Not supported	
PRE13-C	Use the Standard predefined macros to test for versions and features.	Not supported	
PRE30-C	Do not create a universal character name through concatenation	Not supported	
PRE31-C	Avoid side effects in arguments to unsafe macros	Supported	<a href="#">666</a> <a href="#">2666</a>
PRE32-C	Do not use preprocessor directives in invocations of function-like macros	Supported	<a href="#">436</a> <a href="#">9501</a>
DCL00-C	Const-qualify immutable objects	Supported	<a href="#">953</a>
DCL01-C	Do not reuse variable names in subscopes	Supported	<a href="#">578</a>
DCL02-C	Use visually distinct identifiers	Partially supported <sup>2</sup>	<a href="#">9046</a>
DCL03-C	Use a static assertion to test the value of a constant expression	Not supported	
DCL04-C	Do not declare more than one variable per declaration	Partially supported <sup>3</sup>	<a href="#">9146</a>

Guideline	Description	Support Level	Diagnostics
DCL05-C	Use typedefs of non-pointer types only	Not supported	
DCL06-C	Use meaningful symbolic constants to represent literal values	Not supported	
DCL07-C	Include the appropriate type information in function declarators	Supported	718 746 936 9074
DCL08-C	Properly encode relationships in constant definitions	Not statically checkable	
DCL09-C	Declare functions that return <code>errno</code> with a return type of <code>errno_t</code>	Not supported	
DCL10-C	Maintain the contract between the writer and caller of variadic functions	Assistance provided <sup>4</sup>	558 719
DCL11-C	Understand the type issues associated with variadic functions	Assistance provided <sup>4</sup>	175 559 2408
DCL12-C	Implement abstract data types using opaque types	Not supported	
DCL13-C	Declare function parameters that are pointers to values not changed by the function as <code>const</code>	Supported	818
DCL15-C	Declare file-scope objects or functions that do not need external linkage as <code>static</code>	Supported	765
DCL16-C	Use "L," not "l," to indicate a long value	Supported	620
DCL17-C	Beware of miscompiled volatile-qualified variables	Not supported	
DCL18-C	Do not begin integer constants with 0 when specifying a decimal value	Supported	9001
DCL19-C	Minimize the scope of variables and functions	Partially supported	765 9003
DCL20-C	Explicitly specify <code>void</code> when a function accepts no arguments	Partially supported	937
DCL21-C	Understand the storage of compound literals	Not supported	
DCL22-C	Use <code>volatile</code> for data that cannot be cached	Not supported	
DCL23-C	Guarantee that mutually visible identifiers are unique	Supported	621
DCL30-C	Declare objects with appropriate storage durations	Partially supported	604 674 733 789
DCL31-C	Declare identifiers before using them	Supported	601 718 746 808
DCL36-C	Do not declare an identifier with conflicting linkage classifications	Supported	401 839 1051
DCL37-C	Do not declare or define a reserved identifier	Partially supported	978 9071 9093
DCL38-C	Use the correct syntax when declaring a flexible array member	Supported	9040
DCL39-C	Avoid information leakage when passing a structure across a trust boundary	Not supported	
DCL40-C	Do not create incompatible declarations of the same function or object	Supported	18 621 793 4376
DCL41-C	Do not declare variables inside a switch statement before the first case label	Assistance provided	527
EXP00-C	Use parentheses for precedence of operation	Supported	9050
EXP02-C	Be aware of the short-circuit behavior of the logical AND and OR operators	Supported	9007
EXP03-C	Do not assume the size of a structure is the sum of the sizes of its members	Not supported	
EXP05-C	Do not cast away a <code>const</code> qualification	Partially supported	9005
EXP07-C	Do not diminish the benefits of constants by assuming their values in expressions	Not supported	

Guideline	Description	Support Level	Diagnostics
EXP08-C	Ensure pointer arithmetic is used correctly	Partially supported	<a href="#">416</a>
EXP09-C	Use sizeof to determine the size of a type or variable	Not supported	
EXP10-C	Do not depend on the order of evaluation of subexpressions or the order in which side effects take place	Partially supported	<a href="#">564</a> <a href="#">931</a>
EXP11-C	Do not make assumptions regarding the layout of structures with bit-fields	Not supported	
EXP12-C	Do not ignore values returned by functions	Supported	<a href="#">534</a>
EXP13-C	Treat relational and equality operators as if they were nonassociative	Supported	<a href="#">503</a> <a href="#">731</a>
EXP14-C	Beware of integer promotion when performing bitwise operations on integer types smaller than int	Not supported	
EXP15-C	Do not place a semicolon on the same line as an if, for, or while statement	Partially supported <sup>5</sup>	<a href="#">721</a> <a href="#">722</a>
EXP16-C	Do not compare function pointers to constant values	Partially supported <sup>6</sup>	<a href="#">2440</a> <a href="#">2441</a>
EXP19-C	Use braces for the body of an if, for, or while statement	Supported	<a href="#">9012</a>
EXP20-C	Perform explicit tests to determine success, true and false, and equality	Partially supported <sup>7</sup>	<a href="#">697</a>
EXP30-C	Do not depend on the order of evaluation for side effects	Partially supported	<a href="#">564</a>
EXP32-C	Do not access a volatile object through a non-volatile reference	Not supported	
EXP33-C	Do not read uninitialized memory	Supported	<a href="#">530</a> <a href="#">603</a> <a href="#">644</a> <a href="#">901</a>
EXP34-C	Do not dereference null pointers	Partially supported	<a href="#">413</a> <a href="#">418</a> <a href="#">444</a> <a href="#">613</a> <a href="#">668</a>
EXP35-C	Do not modify objects with temporary lifetime	Not supported	
EXP36-C	Do not cast pointers into more strictly aligned pointer types	Partially supported <sup>8</sup>	<a href="#">2445</a>
EXP37-C	Call functions with the correct number and type of arguments	Not supported	
EXP39-C	Do not access a variable through a pointer of an incompatible type	Not supported	
EXP40-C	Do not modify constant objects	Not supported	
EXP42-C	Do not compare padding data	Assistance provided <sup>9</sup>	<a href="#">958</a> <a href="#">959</a>
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	Assistance provided <sup>10</sup>	<a href="#">586</a>
EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic	Partially supported <sup>11</sup>	<a href="#">9006</a>
EXP45-C	Do not perform assignments in selection statements	Partially supported <sup>12</sup>	<a href="#">720</a>
EXP46-C	Do not use a bitwise operator with a Boolean-like operand	Supported	<a href="#">514</a>
EXP47-C	Do not call va_arg with an argument of the incorrect type	Assistance provided <sup>13</sup>	<a href="#">917</a>
INT00-C	Understand the data model used by your implementation(s)	Assistance provided <sup>14</sup>	<a href="#">559</a> <a href="#">705</a> <a href="#">706</a> <a href="#">2403</a>
INT01-C	Use rsize_t or size_t for all integer values representing the size of an object	Not supported	

Guideline	Description	Support Level	Diagnostics
INT02-C	Understand integer conversion rules	Partially supported	<a href="#">501 502 569 570</a> <a href="#">573 574 701 702</a> <a href="#">732 734 737</a>
INT04-C	Enforce limits on integer values originating from tainted sources	Not supported	
INT05-C	Do not use input functions to convert character data if they cannot handle all possible inputs	Supported	<a href="#">586</a>
INT07-C	Use only explicitly signed or unsigned char type for numeric values	Supported	<a href="#">9112</a>
INT08-C	Verify that all integer values are in range	Partially supported	<a href="#">648 650 679 680</a> <a href="#">776 952 2704</a>
INT09-C	Ensure enumeration constants map to unique values	Partially supported	<a href="#">488 9148</a>
INT10-C	Do not assume a positive remainder when using the % operator	Not supported	
INT12-C	Do not make assumptions about the type of a plain int bit-field when used in an expression	Supported	<a href="#">846</a>
INT13-C	Use bitwise operators only on unsigned operands	Partially supported <sup>15</sup>	<a href="#">9233</a>
INT14-C	Avoid performing bitwise and arithmetic operations on the same data	Not supported	
INT15-C	Use intmax_t or uintmax_t for formatted IO on programmer-defined integer types	Not supported	
INT16-C	Do not make assumptions about representation of signed integers	Partially supported <sup>16</sup>	<a href="#">502 2704 9088</a>
INT17-C	Define integer constants in an implementation-independent manner	Not supported	
INT18-C	Evaluate integer expressions in a larger size before comparing or assigning to that size	Not supported	
INT30-C	Ensure that unsigned integer operations do not wrap	Not supported	
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	Not supported	
INT32-C	Ensure that operations on signed integers do not result in overflow	Not supported	
INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors	Not supported	
INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.	Not supported	
INT35-C	Use correct integer precisions	Not supported	
INT36-C	Converting a pointer to integer or integer to pointer	Partially supported <sup>17</sup>	<a href="#">4287</a>
FLP00-C	Understand the limitations of floating-point numbers	Partially supported	<a href="#">777 9252</a>
FLP01-C	Take care in rearranging floating-point expressions	Not statically checkable	
FLP02-C	Avoid using floating-point numbers when precise computation is needed	Partially supported	<a href="#">777 9252</a>
FLP03-C	Detect and handle floating-point errors	Assistance provided	<a href="#">736 9120 9227</a>
FLP04-C	Check floating-point inputs for exceptional values	Not supported	

Guideline	Description	Support Level	Diagnostics
FLP05-C	Do not use denormalized numbers	Not supported	
FLP06-C	Convert integers to floating point for floating-point operations	Supported	<a href="#">653 790 942</a>
FLP07-C	Cast the return value of a function that returns a floating-point type	Not supported	
FLP30-C	Do not use floating-point variables as loop counters	Supported	<a href="#">9009</a>
FLP32-C	Prevent or detect domain and range errors in math functions	Partially supported <sup>18</sup>	<a href="#">2423</a>
FLP34-C	Ensure that floating-point conversions are within range of the new type	Partially supported	<a href="#">735 736 915 922 9118 9227</a>
FLP36-C	Preserve precision when converting integral values to floating-point type	Partially supported	<a href="#">915 922</a>
FLP37-C	Do not use object representations to compare floating-point values	Supported	<a href="#">2498 2499</a>
ARR00-C	Understand how arrays work	Partially supported <sup>19</sup>	<a href="#">409 413 429 613</a>
ARR01-C	Do not apply the sizeof operator to a pointer when taking the size of an array	Supported	<a href="#">682 882</a>
ARR02-C	Explicitly specify array bounds, even if implicitly defined by an initializer	Partially supported	<a href="#">576</a>
ARR30-C	Do not form or use out-of-bounds pointers or array subscripts	Supported	<a href="#">413 415 416 613 661 662 676</a>
ARR32-C	Ensure size arguments for variable length arrays are in a valid range	Assistance provided	<a href="#">9035</a>
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array	Not supported	
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	Partially supported	<a href="#">2662</a>
ARR38-C	Guarantee that library functions do not form invalid pointers	Partially supported	<a href="#">419 420</a>
ARR39-C	Do not add or subtract a scaled integer to a pointer	Not supported	
STR00-C	Represent characters using an appropriate type	Not statically checkable	
STR01-C	Adopt and implement a consistent plan for managing strings	Not statically checkable	
STR02-C	Sanitize data passed to complex subsystems	Not supported	
STR03-C	Do not inadvertently truncate a string	Not supported	
STR04-C	Use plain char for characters in the basic character set	Not supported	
STR05-C	Use pointers to const when referring to string literals	Supported	<a href="#">1776</a>
STR06-C	Do not assume that strtok() leaves the parse string unchanged	Not supported	
STR07-C	Use the bounds-checking interfaces for string manipulation	Supported	<a href="#">586</a>
STR09-C	Don't assume numeric values for expressions with type plain character	Supported	<a href="#">9209</a>
STR10-C	Do not concatenate different type of string literals	Supported	<a href="#">707</a>
STR11-C	Do not specify the bound of a character array initialized with a string literal	Partially supported	<a href="#">784</a>

Guideline	Description	Support Level	Diagnostics
STR30-C	Do not attempt to modify string literals	Partially supported	<a href="#">489</a> <a href="#">1776</a>
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator	Partially supported	<a href="#">421</a> <a href="#">498</a>
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string	Not supported	
STR34-C	Cast characters to unsigned char before converting to larger integer sizes	Partially supported	<a href="#">571</a>
STR37-C	Arguments to character-handling functions must be representable as an unsigned char	Not supported	
STR38-C	Do not confuse narrow and wide character strings and functions	Partially supported <sup>20</sup>	<a href="#">2454</a> <a href="#">2480</a> <a href="#">2481</a>
MEM00-C	Allocate and free memory in the same module, at the same level of abstraction	Partially supported	<a href="#">449</a> <a href="#">2434</a>
MEM01-C	Store a new value in pointers immediately after free()	Not supported	
MEM02-C	Immediately cast the result of a memory allocation function call into a pointer to the allocated type	Assistance provided <sup>21</sup>	<a href="#">908</a>
MEM03-C	Clear sensitive information stored in reusable resources	Not supported	
MEM04-C	Beware of zero-length allocations	Not supported	
MEM05-C	Avoid large stack allocations	Partially supported <sup>22</sup>	<a href="#">9035</a> <a href="#">9070</a>
MEM06-C	Ensure that sensitive data is not written out to disk	Not supported	
MEM07-C	Ensure that the arguments to calloc(), when multiplied, do not wrap	Not supported	
MEM10-C	Define and use a pointer validation function	Not supported	
MEM11-C	Do not assume infinite heap space	Assistance provided <sup>23</sup>	<a href="#">413</a> <a href="#">613</a>
MEM12-C	Consider using a goto chain when leaving a function on error when using and releasing resources	Assistance provided	<a href="#">429</a>
MEM30-C	Do not access freed memory	Supported	<a href="#">449</a> <a href="#">2434</a>
MEM31-C	Free dynamically allocated memory when no longer needed	Supported	<a href="#">429</a>
MEM33-C	Allocate and copy structures containing a flexible array member dynamically	Not supported	
MEM34-C	Only free memory allocated dynamically	Supported	<a href="#">424</a> <a href="#">673</a>
MEM35-C	Allocate sufficient memory for an object	Partially supported	<a href="#">433</a> <a href="#">826</a>
MEM36-C	Do not modify the alignment of objects by calling realloc()	Not supported	
FIO01-C	Be careful using functions that use file names for identification	Not supported	
FIO02-C	Canonicalize path names originating from tainted sources	Not supported	
FIO03-C	Do not make assumptions about fopen() and file creation	Not supported	
FIO05-C	Identify files using multiple file attributes	Not supported	
FIO06-C	Create files with appropriate access permissions	Not supported	

Guideline	Description	Support Level	Diagnostics
FIO08-C	Take care when calling remove() on an open file	Not supported	
FIO09-C	Be careful with binary data when transferring data across systems	Not supported	
FIO10-C	Take care when using the rename() function	Not supported	
FIO11-C	Take care when specifying the mode parameter of fopen()	Supported	<a href="#">2472</a> <a href="#">2473</a>
FIO13-C	Never push back anything other than one read character	Supported	<a href="#">2470</a>
FIO14-C	Understand the difference between text mode and binary mode with file streams	Not supported	
FIO15-C	Ensure that file operations are performed in a secure directory	Not supported	
FIO17-C	Do not rely on an ending null character when using fread()	Not supported	
FIO18-C	Never expect fwrite() to terminate the writing process at a null character	Not supported	
FIO19-C	Do not use fseek() and ftell() to compute the size of a regular file	Not supported	
FIO20-C	Avoid unintentional truncation when using fgets() or fgetws()	Not supported	
FIO21-C	Do not create temporary files in shared directories	Not supported	
FIO22-C	Close files before spawning processes	Not supported	
FIO23-C	Do not exit with unflushed data in stdout or stderr	Not supported	
FIO24-C	Do not open a file that is already open	Not supported	
FIO30-C	Exclude user input from format strings	Partially supported <sup>24</sup>	<a href="#">592</a>
FIO32-C	Do not perform operations on devices that are only appropriate for files	Not supported	
FIO34-C	Distinguish between characters read from a file and EOF or WEOF	Not supported	
FIO37-C	Do not assume that fgets() or fgetws() returns a nonempty string when successful	Not supported	
FIO38-C	Do not copy a FILE object	Partially supported <sup>25</sup>	<a href="#">9047</a>
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call	Supported	<a href="#">2478</a> <a href="#">2479</a>
FIO40-C	Reset strings on fgets() or fgetws() failure	Not supported	
FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	Not supported	
FIO42-C	Close files when they are no longer needed	Partially supported	<a href="#">429</a>
FIO44-C	Only use values for fsetpos() that are returned from fgetpos()	Not supported	
FIO45-C	Avoid TOCTOU race conditions while accessing files	Not supported	
FIO46-C	Do not access a closed file	Supported	<a href="#">2471</a>
FIO47-C	Use valid format strings	Supported	<a href="#">492</a> <a href="#">493</a> <a href="#">494</a> <a href="#">499</a> <a href="#">557</a> <a href="#">558</a> <a href="#">559</a> <a href="#">566</a> <a href="#">705</a> <a href="#">706</a> <a href="#">719</a> <a href="#">816</a> <a href="#">855</a> <a href="#">2401</a> <a href="#">2402</a> <a href="#">2403</a> <a href="#">2404</a> <a href="#">2405</a> <a href="#">2406</a> <a href="#">2407</a>



Guideline	Description	Support Level	Diagnostics
ENV01-C	Do not make assumptions about the size of an environment variable	Supported	<a href="#">669</a>
ENV02-C	Beware of multiple environment variables with the same effective name	Not supported	
ENV03-C	Sanitize the environment when invoking external programs	Not supported	
ENV30-C	Do not modify the object referenced by the return value of certain functions	Not supported	
ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it	Not supported	
ENV32-C	All exit handlers must return normally	Not supported	
ENV33-C	Do not call <code>system()</code>	Supported	<a href="#">586</a>
ENV34-C	Do not store pointers returned by certain functions	Not supported	
SIG00-C	Mask signals handled by noninterruptible signal handlers	Assistance provided <sup>26</sup>	<a href="#">586</a>
SIG01-C	Understand implementation-specific details regarding signal handler persistence	Assistance provided <sup>26</sup>	<a href="#">586</a>
SIG02-C	Avoid using signals to implement normal functionality	Assistance provided <sup>26</sup>	<a href="#">586</a>
SIG30-C	Call only asynchronous-safe functions within signal handlers	Supported	<a href="#">2670</a> <a href="#">2761</a>
SIG31-C	Do not access shared objects in signal handlers	Supported	<a href="#">2765</a>
SIG34-C	Do not call <code>signal()</code> from within interruptible signal handlers	Supported	<a href="#">2762</a> <a href="#">2763</a>
SIG35-C	Do not return from a computational exception signal handler	Supported	<a href="#">2671</a> <a href="#">2764</a>
ERR00-C	Adopt and implement a consistent and comprehensive error-handling policy	Not statically checkable	
ERR01-C	Use <code>ferror()</code> rather than <code>errno</code> to check for FILE stream errors	Not supported	
ERR02-C	Avoid in-band error indicators	Not statically checkable	
ERR03-C	Use runtime-constraint handlers when calling the bounds-checking interfaces	Not supported	
ERR04-C	Choose an appropriate termination strategy	Supported	<a href="#">586</a>
ERR05-C	Application-independent code should provide error detection without dictating error handling	Not supported	
ERR06-C	Understand the termination behavior of <code>assert()</code> and <code>abort()</code>	Supported	<a href="#">586</a>
ERR07-C	Prefer functions that support error checking over equivalent functions that don't	Supported	<a href="#">586</a>
ERR30-C	Set <code>errno</code> to zero before calling a library function known to set <code>errno</code> , and check <code>errno</code> only after the function returns a value indicating failure	Not supported	
ERR32-C	Do not rely on indeterminate values of <code>errno</code>	Not supported	
ERR33-C	Detect and handle standard library errors	Partially supported	<a href="#">534</a>
ERR34-C	Detect errors when converting a string to a number	Assistance provided	<a href="#">586</a>
API00-C	Functions should validate their parameters	Partially supported <sup>27</sup>	<a href="#">413</a> <a href="#">613</a> <a href="#">668</a>
API01-C	Avoid laying out strings in memory directly before sensitive data	Not supported	



Guideline	Description	Support Level	Diagnostics
API02-C	Functions that read or write to or from an array should take an argument to specify the source or target size	Not supported	
API03-C	Create consistent interfaces and capabilities across related functions	Not statically checkable	
API04-C	Provide a consistent and usable error-checking mechanism	Not statically checkable	
API05-C	Use conformant array parameters	Not supported	
API07-C	Enforce type safety	Not supported	
API08-C	Avoid parameter names in a function prototype	Not supported	
API09-C	Compatible values should have the same type	Partially supported	<a href="#">737</a>
API10-C	APIs should have security options enabled by default	Not supported	
MSC00-C	Compile cleanly at high warning levels	Not supported	
MSC01-C	Strive for logical completeness	Partially supported	<a href="#">474</a> <a href="#">744</a> <a href="#">787</a> <a href="#">9013</a>
MSC04-C	Use comments consistently and in a readable fashion	Supported	<a href="#">1</a> <a href="#">427</a> <a href="#">602</a> <a href="#">689</a> <a href="#">853</a> <a href="#">9059</a> <a href="#">9060</a> <a href="#">9066</a> <a href="#">9259</a>
MSC05-C	Do not manipulate time_t typed values directly	Not supported	
MSC06-C	Beware of compiler optimizations	Assistance provided	<a href="#">586</a>
MSC09-C	Character encoding: Use subset of ASCII for safety	Not supported	
MSC10-C	Character encoding: UTF8-related issues	Not supported	
MSC11-C	Incorporate diagnostic tests using assertions	Not supported	
MSC12-C	Detect and remove code that has no effect or is never executed	Supported	<a href="#">438</a> <a href="#">474</a> <a href="#">505</a> <a href="#">522</a> <a href="#">523</a> <a href="#">527</a> <a href="#">528</a> <a href="#">529</a> <a href="#">563</a> <a href="#">612</a> <a href="#">714</a> <a href="#">715</a> <a href="#">719</a> <a href="#">749</a> <a href="#">750</a> <a href="#">751</a> <a href="#">752</a> <a href="#">753</a> <a href="#">754</a> <a href="#">755</a> <a href="#">756</a> <a href="#">757</a> <a href="#">758</a> <a href="#">768</a> <a href="#">769</a> <a href="#">774</a> <a href="#">827</a> <a href="#">838</a> <a href="#">1972</a>
MSC13-C	Detect and remove unused values	Partially supported	<a href="#">438</a> <a href="#">505</a> <a href="#">529</a> <a href="#">715</a> <a href="#">838</a>
MSC14-C	Do not introduce unnecessary platform dependencies	Not supported <sup>28</sup>	
MSC15-C	Do not depend on undefined behavior	Not supported <sup>29</sup>	
MSC17-C	Finish every set of statements associated with a case label with a break statement	Supported	<a href="#">616</a> <a href="#">825</a>
MSC18-C	Be careful while handling sensitive data, such as passwords, in program code	Partially supported <sup>30</sup>	<a href="#">586</a>
MSC19-C	For functions that return an array, prefer returning an empty array over a null value	Partially supported	<a href="#">413</a> <a href="#">418</a> <a href="#">419</a> <a href="#">420</a> <a href="#">473</a> <a href="#">613</a> <a href="#">661</a> <a href="#">662</a> <a href="#">668</a> <a href="#">669</a> <a href="#">670</a>
MSC20-C	Do not use a switch statement to transfer control into a complex block	Supported	<a href="#">646</a> <a href="#">9055</a>
MSC21-C	Use robust loop termination conditions	Partially supported	<a href="#">440</a> <a href="#">442</a> <a href="#">443</a> <a href="#">444</a> <a href="#">445</a> <a href="#">2650</a>
MSC22-C	Use the setjmp(), longjmp() facility securely	Not supported	
MSC23-C	Beware of vendor-specific library and language differences	Not supported	

Guideline	Description	Support Level	Diagnostics
MSC24-C	Do not use deprecated or obsolescent functions	Supported	<a href="#">586</a>
MSC30-C	Do not use the rand() function for generating pseudorandom numbers	Supported	<a href="#">586</a>
MSC32-C	Properly seed pseudorandom number generators	Supported	<a href="#">2460</a> <a href="#">2461</a> <a href="#">2760</a>
MSC33-C	Do not pass invalid data to the asctime() function	Supported	<a href="#">586</a>
MSC37-C	Ensure that control never reaches the end of a non-void function	Supported	<a href="#">533</a>
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	Not supported	
MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	Not supported	
MSC40-C	Do not violate constraints	Not supported <sup>31</sup>	
MSC41-C	Never hard code sensitive information	Assistance provided <sup>32</sup>	<a href="#">2460</a>
WIN00-C	Be specific when dynamically loading libraries	Supported	<a href="#">586</a>
WIN01-C	Do not forcibly terminate execution	Supported	<a href="#">586</a>
WIN02-C	Restrict privileges when spawning child processes	Supported	<a href="#">586</a>
WIN03-C	Understand HANDLE inheritance	Not supported	
WIN04-C	Consider encrypting function pointers	Not supported	
WIN30-C	Properly pair allocation and deallocation functions	Not supported	
CON01-C	Acquire and release synchronization primitives in the same module, at the same level of abstraction	Partially supported <sup>33</sup>	<a href="#">454</a> <a href="#">455</a> <a href="#">456</a>
CON02-C	Do not use volatile as a synchronization primitive	Not supported	
CON03-C	Ensure visibility when accessing shared variables	Not supported	
CON04-C	Join or detach threads even if their exit status is unimportant	Not supported	
CON05-C	Do not perform operations that can block while holding a lock	Not supported	
CON06-C	Ensure that every mutex outlives the data it protects	Not supported	
CON07-C	Ensure that compound operations on shared variables are atomic	Not supported	
CON08-C	Do not assume that a group of calls to independently atomic methods is atomic	Not supported	
CON09-C	Avoid the ABA problem when using lock-free algorithms	Not supported	
CON30-C	Clean up thread-specific storage	Not supported	
CON31-C	Do not destroy a mutex while it is locked	Not supported	
CON32-C	Prevent data races when accessing bit-fields from multiple threads	Partially supported <sup>34</sup>	<a href="#">457</a>
CON33-C	Avoid race conditions when using library functions	Supported	<a href="#">586</a>
CON34-C	Declare objects shared between threads with appropriate storage durations	Not supported	
CON35-C	Avoid deadlock by locking in a predefined order	Supported	<a href="#">2462</a>
CON36-C	Wrap functions that can spuriously wake up in a loop	Not supported	

Guideline	Description	Support Level	Diagnostics
CON37-C	Do not call <code>signal()</code> in a multithreaded program	Supported	<a href="#">586</a>
CON38-C	Preserve thread safety and liveness when using condition variables	Not supported	
CON39-C	Do not join or detach a thread that was previously joined or detached	Not supported	
CON40-C	Do not refer to an atomic variable twice in an expression	Not supported	
CON41-C	Wrap functions that can fail spuriously in a loop	Not supported	
CON42-C	Don't allow attackers to influence environment variables that control concurrency parameters	Not supported	
CON43-C	Do not allow data races in multithreaded code	Partially supported <sup>35</sup>	<a href="#">457</a>
POS01-C	Check for the existence of links when dealing with files	Not supported	
POS02-C	Follow the principle of least privilege	Not supported	
POS04-C	Avoid using <code>PTHREAD_MUTEX_NORMAL</code> type mutex locks	Supported	<a href="#">586</a>
POS05-C	Limit access to files by creating a jail	Not supported	
POS30-C	Use the <code>readlink()</code> function properly	Not supported	
POS33-C	Do not use <code>vfork()</code>	Supported	<a href="#">586</a>
POS34-C	Do not call <code>putenv()</code> with a pointer to an automatic variable as the argument	Supported	<a href="#">2601</a>
POS35-C	Avoid race conditions while checking for the existence of a symbolic link	Not supported	
POS36-C	Observe correct revocation order while relinquishing privileges	Not supported	
POS37-C	Ensure that privilege relinquishment is successful	Not supported	
POS38-C	Beware of race conditions when using <code>fork</code> and file descriptors	Not supported	
POS39-C	Use the correct byte ordering when transferring data between systems	Not supported	
POS44-C	Do not use signals to terminate threads	Supported	<a href="#">586</a>
POS47-C	Do not use threads that can be canceled asynchronously	Supported	<a href="#">586</a>
POS48-C	Do not unlock or destroy another POSIX thread's mutex	Not supported	
POS49-C	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed	Partially supported <sup>36</sup>	<a href="#">457</a>
POS50-C	Declare objects shared between POSIX threads with appropriate storage durations	Not supported	
POS51-C	Avoid deadlock with POSIX threads by locking in predefined order	Not supported	
POS52-C	Do not perform operations that can block while holding a POSIX lock	Not supported	
POS53-C	Do not use more than one mutex for concurrent waiting operations on a condition variable	Not supported	
POS54-C	Detect and handle POSIX library errors	Assistance provided	<a href="#">413</a> <a href="#">534</a> <a href="#">613</a>

<sup>1</sup>Reports any use of pasting or stringizing operators in a macro definition<sup>2</sup>Does not report 'Q' or 'D' vs '0' or 'O'<sup>3</sup>Exceptions not supported

- <sup>4</sup>Reports issues involving format strings
- <sup>5</sup>Reports missing body from if, for, or while with semi-colon immediately following predicate
- <sup>6</sup>Reports address of function, array, or variable directly or indirectly compared to null
- <sup>7</sup>Reports comparisons of Boolean values to constants other than 0
- <sup>8</sup>Reports casts directly from a pointer to a less strictly aligned type to a pointer to a more strictly aligned type
- <sup>9</sup>Reports structures which require padding between members or after the last member
- <sup>10</sup>Reports use of the restrict keyword
- <sup>11</sup>Reports use of sizeof with an expression that would have side effects
- <sup>12</sup>Reports Boolean test of unparenthesized assignment
- <sup>13</sup>Reports argument promotion to match prototype
- <sup>14</sup>Reports data type inconsistencies in format strings
- <sup>15</sup>Reports use of a bitwise operator on an expression with a signed MISRA C 2004 underlying type
- <sup>16</sup>Reports bitwise not of signed quantity, declaration of named signed single-bit bitfields, and negation of the minimum negative integer
- <sup>17</sup>Reports casts from pointer types to smaller integer types which lose information
- <sup>18</sup>Reports domain errors for functions with the Semantics `*dom_1`, `*dom_lt0`, or `*dom_lt1`, including standard library math functions
- <sup>19</sup>Conceptually includes all other ARR items which are mapped to their respective guidelines; explicit mappings for ARR00 are present when a situation mentioned in the guideline itself is encountered
- <sup>20</sup>Reports illegal conversions involving pointers to `char` or `wchar_t` as well as byte/wide-oriented stream inconsistencies
- <sup>21</sup>Reports implicit conversions from `void*` to another type
- <sup>22</sup>Reports use of variable length arrays and recursion
- <sup>23</sup>Reports use of null pointers including those which could be returned when a call to an allocation function fails
- <sup>24</sup>Reports non-literal format strings
- <sup>25</sup>Reports when a `FILE` pointer is dereferenced
- <sup>26</sup>Reports use of the signal function
- <sup>27</sup>Reports use of null pointers including function parameters which are assumed to have the potential to be null
- <sup>28</sup>Many instances of platform-dependent behavior are reported but do not include appended text citing this guideline
- <sup>29</sup>Many instances of undefined behavior are reported but do not include appended text citing this guideline
- <sup>30</sup>Reports functions that read passwords from the user or that take a password as an argument instead of prompting the user as well as insecure password erasure (see also MSC41-C)
- <sup>31</sup>Many constraint violations are reported as error messages but do not include appended text citing this guideline
- <sup>32</sup>Reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate
- <sup>33</sup>Acquire and release synchronization primitives within the same scope
- <sup>34</sup>Access is detected at the object level (not at the field level)
- <sup>35</sup>Access is detected at the object level (not at the field level)
- <sup>36</sup>Access is detected at the object level (not at the field level)

## 15 CWE™ Checking

### 15.1 Introduction to CWE Support

Common Weakness Enumeration (CWE) is a community-developed list of common software and hardware weakness types that have security ramifications. A “weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. The CWE List and associated classification taxonomy serve as a language that can be used to identify and describe these weaknesses in terms of CWEs.

PC-lint Plus provides support for detecting C/C++ software issues described by the CWE (version 4.12 as of January 3, 2024) using the `au-cwe.lnt` author file which is distributed with the product in the `lnt/` directory. The author file include `-append` options which cause messages that are used to report CWE weaknesses to be annotated with the corresponding CWE identifier. In addition, the CWE Coverage Claim Representation (CCR) XML document (`au-cwe.xml`) is in the `lnt/` directory. All weaknesses in the following mappings and views are included:

- CWE Top 25 (2023-2019)
- Software Written in C
- Software Written in C++
- SEI CERT C Coding Standard
- The CERT C Secure Coding Standard (2008)
- SEI CERT C++ Coding Standard (2016)

The author file enables checks for both library and non-library code. This means that the standard headers employed by your source code are subject to the same scrutiny as the rest of the project. This is often a project requirement but can result in a lot of noise if library code is not subject to the same compliance requirements as the rest of the project. The simplest way to disable CWE checks for library code is to place the options `-wlib(4)` `-wlib(1)` immediately after the author file is referenced. This raises and immediately lowers the warning level for libraries resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.

The following subsections document the level of support provided by PC-lint Plus for the CWE. The weakness identifier, level of support, and primary enforcing messages are provided. While every effort is made to ensure the correctness of the information provided here, Vector Informatik GmbH makes no guarantee with respect to the accuracy of the information conveyed.

The following terms are used to characterize the support that PC-lint Plus provides each weakness:

- **Supported** - For statically checkable weaknesses, the weakness is comprehensively supported and no false positives nor false negatives are expected. For weaknesses that are not fully statically checkable, substantial support is provided to detect statically checkable issues.
- **Partially Supported** - Meaningful support is provided but there may be cases where false positives and/or false negatives may occur due to limitations in the currently implemented detection method, etc.
- **Assistance Provided** - While detection of the actual weakness is not supported (or cannot be statically checked), *potential* instances of the weakness are diagnosed.
- **Not Supported** - No meaningful level of support is currently provided for this weakness.
- **Not Statically Checkable** - Instances of the weakness cannot be detected by means of static analysis.

---

™ CWE is a trademark of MITRE Corporation.

## 15.2 CWE Support

### 15.2.1 CWE Support Summary

Category	# Weaknesses	% of Statically Checkable Weaknesses
Supported	113	60.8%
Comprehensive	27	14.5%
Partial	83	44.6%
Assistance	3	1.6%
Not Supported	73	39.2%
Not Statically Checkable	8	
TOTAL	194	

### 15.2.2 CWE Support Matrix

ID	Name	Support Level	Diagnostics
14	Compiler Removal of Code to Clear Buffers	Assistance provided	<a href="#">586</a>
20	Improper Input Validation	Partially supported	<a href="#">586</a> <a href="#">592</a>
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Not supported	
37	Path Traversal: '/absolute/pathname/here'	Not supported	
38	Path Traversal: '\\absolute\\pathname\\here'	Not supported	
39	Path Traversal: 'C:dirname'	Not supported	
41	Improper Resolution of Path Equivalence	Not supported	
59	Improper Link Resolution Before File Access ('Link Following')	Not supported	
62	UNIX Hard Link	Not supported	
64	Windows Shortcut Following (.LNK)	Not supported	
65	Windows Hard Link	Not supported	
67	Improper Handling of Windows Device Names	Not supported	
77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	Not supported	
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Partially supported	<a href="#">586</a>
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Not supported	
88	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	Partially supported	<a href="#">586</a>
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Not supported	
94	Improper Control of Generation of Code ('Code Injection')	Not supported	
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Partially supported	<a href="#">409</a> <a href="#">413</a> <a href="#">415</a> <a href="#">416</a> <a href="#">419</a> <a href="#">420</a> <a href="#">421</a> <a href="#">429</a> <a href="#">498</a> <a href="#">613</a> <a href="#">661</a> <a href="#">662</a> <a href="#">669</a> <a href="#">670</a> <a href="#">676</a> <a href="#">2662</a>

ID	Name	Support Level	Diagnostics
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Partially supported	421 498
121	Stack-based Buffer Overflow	Partially supported	419 420 421 498
122	Heap-based Buffer Overflow	Partially supported	421 498
123	Write-what-where Condition	Partially supported	413 415 416 419 420 421 498 613 661 662 676
124	Buffer Underwrite ('Buffer Underflow')	Partially supported	428 676
125	Out-of-bounds Read	Partially supported	413 415 416 419 420 421 498 613 661 662 676
126	Buffer Over-read	Partially supported	413 415 416 419 420 661 662 676
127	Buffer Under-read	Partially supported	413 415 416 420 613 661 662 676
128	Wrap-around Error	Not supported	
129	Improper Validation of Array Index	Partially supported	409 413 415 416 419 420 429 613 661 662 676
130	Improper Handling of Length Parameter Inconsistency	Not supported	
131	Incorrect Calculation of Buffer Size	Partially supported	433 826
134	Use of Externally-Controlled Format String	Partially supported	492 493 494 499 557 558 559 566 592 705 706 719 816 855 905 2401 2402 2403 2404 2405 2406 2407
135	Incorrect Calculation of Multi-Byte String Length	Partially supported	2452 2454
170	Improper Null Termination	Partially supported	495 496 693 784 840
176	Improper Handling of Unicode Encoding	Partially supported	426
188	Reliance on Data/Memory Layout	Not supported	
190	Integer Overflow or Wraparound	Partially supported	433 826
191	Integer Underflow (Wrap or Wraparound)	Not supported	
192	Integer Coercion Error	Partially supported	501 502 569 570 573 574 586 701 702 732 734 737
193	Off-by-one Error	Partially supported	421 498
194	Unexpected Sign Extension	Not supported	
195	Signed to Unsigned Conversion Error	Partially supported	732 737
196	Unsigned to Signed Conversion Error	Not supported	
197	Numeric Truncation Error	Partially supported	501 502 569 570 573 574 586 701 702 732 734 735 736 737 915 922 9118 9227

ID	Name	Support Level	Diagnostics
200	Exposure of Sensitive Information to an Unauthorized Actor	Not supported	
226	Sensitive Information in Resource Not Removed Before Reuse	Not supported	
241	Improper Handling of Unexpected Data Type	Not supported	
242	Use of Inherently Dangerous Function	Supported	421 586
243	Creation of chroot Jail Without Changing Working Directory	Not supported	
244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	Not supported	
248	Uncaught Exception	Partially supported	1559 1560
252	Unchecked Return Value	Partially supported	413 534 613
253	Incorrect Check of Function Return Value	Partially supported	413 534 613
269	Improper Privilege Management	Not supported	
272	Least Privilege Violation	Not supported	
273	Improper Check for Dropped Privileges	Not supported	
276	Incorrect Default Permissions	Not statically checkable	
279	Incorrect Execution-Assigned Permissions	Not supported	
287	Improper Authentication	Not statically checkable	
295	Improper Certificate Validation	Not supported	
306	Missing Authentication for Critical Function	Not supported	
327	Use of a Broken or Risky Cryptographic Algorithm	Supported	586 2460 2461 2760
330	Use of Insufficiently Random Values	Supported	586 2460 2461 2760
331	Insufficient Entropy	Supported	2460 2461 2760
337	Predictable Seed in Pseudo-Random Number Generator (PRNG)	Supported	2461 2760 2960
338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	Supported	586
352	Cross-Site Request Forgery (CSRF)	Not statically checkable	
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	Partially supported	457 459 460 461 462 2511 2512 2513
363	Race Condition Enabling Link Following	Not supported	
364	Signal Handler Race Condition	Partially supported	2670 2761 2762 2763 2765
366	Race Condition within a Thread	Partially supported	457
367	Time-of-check Time-of-use (TOCTOU) Race Condition	Not supported	
369	Divide By Zero	Partially supported	54 414 736 9120 9227
374	Passing Mutable Objects to an Untrusted Method	Not supported	
375	Returning a Mutable Object to an Untrusted Caller	Not supported	
377	Insecure Temporary File	Supported	586
379	Creation of Temporary File in Directory with Insecure Permissions	Not supported	
391	Unchecked Error Condition	Partially supported	413 534 586 613 2423



ID	Name	Support Level	Diagnostics
396	Declaration of Catch for Generic Exception	Partially supported	<a href="#">1766</a> <a href="#">1966</a>
397	Declaration of Throws for Generic Exception	Not supported	
398	Indicator of Poor Code Quality	Partially supported	<a href="#">568</a> <a href="#">620</a> <a href="#">687</a> <a href="#">691</a> <a href="#">692</a> <a href="#">721</a> <a href="#">722</a> <a href="#">775</a> <a href="#">777</a> <a href="#">840</a> <a href="#">9063</a> <a href="#">9138</a>
400	Uncontrolled Resource Consumption	Not supported	
401	Missing Release of Memory after Effective Lifetime	Supported	<a href="#">429</a>
403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')	Partially supported	<a href="#">429</a>
404	Improper Resource Shutdown or Release	Partially supported	<a href="#">429</a>
415	Double Free	Partially supported	<a href="#">429</a> <a href="#">449</a> <a href="#">2434</a>
416	Use After Free	Partially supported	<a href="#">449</a> <a href="#">2434</a>
426	Untrusted Search Path	Not supported	
434	Unrestricted Upload of File with Dangerous Type	Not supported	
456	Missing Initialization of a Variable	Partially supported	<a href="#">738</a> <a href="#">1401</a> <a href="#">1539</a>
457	Use of Uninitialized Variable	Supported	<a href="#">530</a> <a href="#">603</a> <a href="#">644</a> <a href="#">901</a> <a href="#">1416</a>
459	Incomplete Cleanup	Partially supported	<a href="#">429</a>
460	Improper Cleanup on Thrown Exception	Not supported	
462	Duplicate Key in Associative List (Alist)	Assistance provided <sup>1</sup>	<a href="#">586</a>
463	Deletion of Data Structure Sentinel	Partially supported	<a href="#">496</a> <a href="#">693</a> <a href="#">840</a>
464	Addition of Data Structure Sentinel	Partially supported	<a href="#">495</a> <a href="#">784</a>
466	Return of Pointer Value Outside of Expected Range	Not supported	
467	Use of sizeof() on a Pointer Type	Partially supported	<a href="#">433</a> <a href="#">682</a> <a href="#">826</a> <a href="#">882</a>
468	Incorrect Pointer Scaling	Partially supported	<a href="#">416</a>
469	Use of Pointer Subtraction to Determine Size	Not supported	
474	Use of Function with Inconsistent Implementations	Not supported	
476	NULL Pointer Dereference	Partially supported	<a href="#">413</a> <a href="#">418</a> <a href="#">444</a> <a href="#">613</a> <a href="#">668</a>
477	Use of Obsolete Function	Partially supported	<a href="#">586</a>
478	Missing Default Case in Multiple Condition Expression	Supported	<a href="#">744</a>
479	Signal Handler Use of a Non-reentrant Function	Supported	<a href="#">2670</a> <a href="#">2761</a> <a href="#">2762</a> <a href="#">2763</a>
480	Use of Incorrect Operator	Partially supported	<a href="#">514</a> <a href="#">720</a>
481	Assigning instead of Comparing	Partially supported	<a href="#">720</a>
482	Comparing instead of Assigning	Partially supported	<a href="#">522</a>
483	Incorrect Block Delimitation	Partially supported	<a href="#">525</a> <a href="#">539</a> <a href="#">725</a>
484	Omitted Break Statement in Switch	Supported	<a href="#">616</a> <a href="#">825</a>
493	Critical Public Variable Without Final Modifier	Not statically checkable	
495	Private Data Structure Returned From A Public Method	Supported	<a href="#">1535</a> <a href="#">1536</a> <a href="#">1537</a>
496	Public Data Assigned to Private Array-Typed Field	Not supported	

ID	Name	Support Level	Diagnostics
498	Cloneable Class Containing Sensitive Information	Not statically checkable	
500	Public Static Field Not Marked Final	Not supported	
502	Deserialization of Untrusted Data	Not supported	
522	Insufficiently Protected Credentials	Not supported	
528	Exposure of Core Dump File to an Unauthorized Control Sphere	Not supported	
543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context	Not supported	
544	Missing Standardized Error Handling Mechanism	Not statically checkable	
547	Use of Hard-coded, Security-relevant Constants	Not supported	
552	Files or Directories Accessible to External Parties	Not supported	
558	Use of getlogin() in Multithreaded Application	Supported	<a href="#">586</a>
560	Use of umask() with chmod-style Argument	Not supported	
561	Dead Code	Supported	<a href="#">438 474 505 522 523 527 528 529 563 612 681 714 715 719 749 750 751 752 753 754 755 756 757 758 768 769 774 827 838 1972</a>
562	Return of Stack Variable Address	Partially supported	<a href="#">604 674 733 789 2601</a>
563	Assignment to Variable without Use	Not supported	
570	Expression is Always False	Partially supported	<a href="#">685 774 944 948 7774</a>
571	Expression is Always True	Partially supported	<a href="#">685 774 944 948</a>
587	Assignment of a Fixed Address to a Pointer	Partially supported	<a href="#">4287</a>
590	Free of Memory not on the Heap	Partially supported	<a href="#">424 673</a>
591	Sensitive Data Storage in Improperly Locked Memory	Not supported	
597	Use of Wrong Operator in String Comparison	Partially supported	<a href="#">779</a>
606	Unchecked Input for Loop Condition	Not supported	
611	Improper Restriction of XML External Entity Reference	Not supported	
628	Function Call with Incorrectly Specified Arguments	Supported	<a href="#">558 719</a>
662	Improper Synchronization	Partially supported	<a href="#">586 2765</a>
664	Improper Control of a Resource Through its Lifetime	Supported	<a href="#">2478 2479</a>
665	Improper Initialization	Partially supported	<a href="#">576</a>
666	Operation on Resource in Wrong Phase of Lifetime	Supported	<a href="#">449 2434 2471</a>
667	Improper Locking	Partially supported	<a href="#">457 459 460 461 2511</a>
672	Operation on a Resource after Expiration or Release	Supported	<a href="#">449 2434 2471</a>

ID	Name	Support Level	Diagnostics
675	Multiple Operations on Resource in Single-Operation Context	Not supported	
676	Use of Potentially Dangerous Function	Partially supported	421 498 586
680	Integer Overflow to Buffer Overflow	Partially supported	433 826
681	Incorrect Conversion between Numeric Types	Partially supported	735 736 776 790 915 922 9118 9227
682	Incorrect Calculation	Partially supported	2423 9112 9233
684	Incorrect Provision of Specified Functionality	Not supported	
685	Function Call With Incorrect Number of Arguments	Partially supported	118 119 492 493 494 499 557 558 559 566 705 706 719 746 816 855 2401 2402 2403 2404 2405 2406 2407
686	Function Call With Incorrect Argument Type	Partially supported	492 493 494 499 557 558 559 566 705 706 719 816 855 2401 2402 2403 2404 2405 2406 2407 2601
687	Function Call With Incorrectly Specified Argument Value	Partially supported	422 426 432 464 671 1415 2423 2623
688	Function Call With Incorrect Variable or Reference as Argument	Not supported	
689	Permission Race Condition During Resource Copy	Not supported	
690	Unchecked Return Value to NULL Pointer Dereference	Partially supported	413 418 444 613 668
696	Incorrect Behavior Order	Not supported	
697	Incorrect Comparison	Not supported	
704	Incorrect Type Conversion or Cast	Partially supported	571 4287 9005
705	Incorrect Control Flow Scoping	Partially supported	586
732	Incorrect Permission Assignment for Critical Resource	Not supported	
733	Compiler Optimization Removal or Modification of Security-critical Code	Not supported	
754	Improper Check for Unusual or Exceptional Conditions	Not supported	
758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	Partially supported	449 530 533 564 586 603 644 901 2434 2471 4287 9035
762	Mismatched Memory Management Routines	Partially supported	424 673
766	Critical Data Element Declared Public	Not supported	
767	Access to Critical Private Variable via Public Method	Supported	1535 1536
768	Incorrect Short Circuit Evaluation	Supported	9007
771	Missing Reference to Active Allocated Resource	Partially supported	429

ID	Name	Support Level	Diagnostics
772	Missing Release of Resource after Effective Lifetime	Partially supported	<a href="#">423 429 593 672</a>
773	Missing Reference to Active File Descriptor or Handle	Partially supported	<a href="#">429</a>
775	Missing Release of File Descriptor or Handle after Effective Lifetime	Partially supported	<a href="#">429</a>
781	Improper Address Validation in IOCTL with METHOD_NEITHER I/O Control Code	Not supported	
782	Exposed IOCTL with Insufficient Access Control	Not supported	
783	Operator Precedence Logic Error	Supported	<a href="#">9050 9113</a>
785	Use of Path Manipulation Function without Maximum-sized Buffer	Partially supported	<a href="#">426</a>
786	Access of Memory Location Before Start of Buffer	Supported	<a href="#">413 415 416 613 661 662 676</a>
787	Out-of-bounds Write	Supported	<a href="#">413 415 416 613 661 662 676</a>
789	Memory Allocation with Excessive Size Value	Partially supported	<a href="#">433 826</a>
798	Use of Hard-coded Credentials	Assistance provided <sup>2</sup>	<a href="#">2460</a>
805	Buffer Access with Incorrect Length Value	Partially supported	<a href="#">419 420</a>
806	Buffer Access Using Size of Source Buffer	Not supported	
825	Expired Pointer Dereference	Partially supported	<a href="#">449</a>
832	Unlock of a Resource that is not Locked	Partially supported	<a href="#">455 2493</a>
833	Deadlock	Partially supported	<a href="#">2410 2457 2462 2488 2489</a>
839	Numeric Range Comparison Without Minimum Check	Partially supported	<a href="#">676</a>
843	Access of Resource Using Incompatible Type ('Type Confusion')	Partially supported	<a href="#">559 705 706 857 3403</a>
862	Missing Authorization	Not statically checkable	
863	Incorrect Authorization	Not statically checkable	
908	Use of Uninitialized Resource	Supported	<a href="#">530 603 644 901</a>
910	Use of Expired File Descriptor	Supported	<a href="#">2471</a>
911	Improper Update of Reference Count	Not supported	
918	Server-Side Request Forgery (SSRF)	Not supported	
1325	Improperly Controlled Sequential Memory Allocation	Not supported	
1335	Incorrect Bitwise Shift of Integer	Supported	<a href="#">504 572 598</a>
1341	Multiple Releases of Same Resource or Handle	Partially supported	<a href="#">2471</a>

<sup>1</sup>Reports usage of the `std::multimap`, `std::multiset`, `std::unordered_multimap`, and `std::unordered_multiset` classes

<sup>2</sup>Reports when a literal is provided as an argument to a function parameter with the "noliteral" argument Semantic; several Windows API functions are marked as such and the `-sem` option can apply it to other functions as appropriate

## 16 Queries

### 16.1 Introduction

The PC-lint Plus *Query Language* is used to express queries appearing in the `-astquery`, `-equery`, and `+equery` options. The Queries feature provides a facility to extend the functionality of PC-lint Plus by defining custom checks and diagnostics and suppressing messages based on dynamic introspection. AST nodes may be inspected and walked during the execution of queries. Queries are composed of *query expressions* which constitute the statically-typed, domain-specific language described in this chapter.

### 16.2 Overview

While the sections that follow document the details of the Query Language, this section provides a brief overview by means of a couple of practical motivating examples.

A query provided to a `-astquery` option will be evaluated for every AST node in an analyzed module, such queries are typically used to perform custom checks of arbitrary complexity and issue corresponding messages when user-defined criteria are met.

For example, the below `-astquery` option contains a query that will cause a custom message to be issued when a non-static data member of a class, structure, or union has an unsigned integral type and a name that does not begin with "u":

```
-astquery(FieldDecl() : {
    getType().isUnsignedIntegerType()
    hasName() && !getNameAsString().startsWith("u")
    message(8001 "unsigned data member '" currentnode "' missing 'u' prefix")
})
```

The argument to the `-astquery` option comprises the query. The `FieldDecl` function is used to ensure that the rest of the query is evaluated in the context of a field and is only executed for AST nodes representing fields. `getType` yields the type of the field and `isUnsignedIntegerType` is true if the this type is an unsigned integral type, the query terminates if it is not. If the field has a name, the `getNameAsString` function extracts it, if it starts with "u" then the query terminates, otherwise the `message` operator is used to issue a custom message. When this option is used with a module containing:

```
struct X {
    unsigned short us;
    unsigned int ix;
};
```

the message produced will look like:

```
info 8001: unsigned data member 'X::ix' missing 'u' prefix
    unsigned int ix;
    ^
```

The use of `currentnode` causes the message to be parameterized with a symbol corresponding to the field which is automatically rendered as expected and allows the message to be suppressed for specific fields using `-esym` or specific types using `-etype`.

A query may also be used with a `-equery` or `+equery` option to suppress messages via criteria that cannot otherwise be expressed such as by exploration of the AST nodes that correspond to a symbol in a message. A query used with one of these options is evaluated once for every message that PC-lint Plus considers issuing and will `vote` against the issuance of the message if the provided query matches.

For example, message [818](#) (parameter of function could be pointer to const) is parameterized by two symbols, the parameter and the function. The following `-equery` option employs a query to suppress the message when the function symbol was instantiated from a template:

```
-eqquery(818, msgSymbolParam(2).FunctionDecl().isTemplateInstantiation())
```

See the [Examples](#) section for many more examples.

## 16.3 Query Fundamentals

### 16.3.1 Types and Values

Every expression has a static type determined at query parse time. Every type, except for the `Void` type, may either have the special empty value of `None` or contain a value representable by its type. The types supported by Queries are listed in the below table.

Type	Representable Values
ASTNode	A node in the program AST.
ASTType	A type in the program AST.
Boolean	The values <code>true</code> and <code>false</code> .
Floating	Double precision floating point values.
Integral	64-bit signed integer values.
Location	Source code locations.
String	Character strings.
Void	-

In a Boolean context, all values are *truthy* except the Boolean value `false` and a `None` value. Note in particular that 0, 0.0, "", and an empty Location value are all *truthy* values.

For example, if `$x` and `$y` are Integral variables, the expression:

```
$x + $y
```

will yield either an Integral value or `None` (if either `$x` or `$y` is `None`, so is the result of adding them together as the `None` value is generally propagative).

### 16.3.2 Query Execution

A *query* consists of one or more *query expression* components that are executed in order. Evaluation of a *query expression* is typically terminated when a contained sub-expression yields a *falsy* value, if a top-level *query expression* yields a *falsy* value then evaluation of the query terminates. For example:

```
$x = SwitchStmt(currentnode)
$x.numSwitchCases > 100
message($x 8001 "lots of switch cases!")
```

If the above query is executed for a non-switch-statement node, the `SwitchStmt` conversion function will yield an empty value (`None`) which will be assigned to the variable `$x`. When `$x` is used in the call to `numSwitchCases`, the empty value of `$x` will propagate across the expression causing it to evaluate to `None` and the `message` expression will not be evaluated. Otherwise, if the number of switch cases is  $\leq 100$ , the second query expression will yield an empty value and evaluation will terminate before the message expression is reached. If the message expression is reached, the message is issued and evaluation would continue if there were any other query expressions following it.

In most cases, the evaluation of an empty value will propagate through enclosing expressions causing them to evaluate as `None`. Situations where this does not occur include cases where the empty value appears:

- in the predicate of a control statement (e.g. `if (IfStmt) { ... }`, evaluation will unconditionally continue after the `if` statement).
- as the operand to the logical `!` operator.

- in the LHS of a logical **or** operator.
- as an argument to the **str**, **message**, or **verbosify** operators.

Query expressions are self-terminating, expressions are not delimited with a special character. In the previous example, the entire query could have been written on a single line without changing the behavior.

### 16.3.3 Numeric Operators

#### 16.3.3.1 Multiplicative and Additive Operators

The standard multiplicative operators (**\***, **/**, and **%**) and additive operators (**+** and **-**) are available for use with numeric types. When used with two Integral types, the result of the operation is Integral. When at least one operand has Floating type, the result has Floating type. When one operand is Integral and the other is Floating, the Integral operand is implicitly converted to a Floating type. The unary negation operator (**-**) may be used with Integral and Floating types. All of the numeric operators yield a **None** value if any of the provided operands is **None**. The **/** and **%** operators will also yield a **None** value if the RHS operand is zero.

#### 16.3.3.2 Bitwise Operators

The binary infix bitwise operators **&** and **|** accept two Integral operands and yield the bitwise AND and bitwise OR result, respectively. The unary prefix complement operator **~** accepts an Integral operand and yields the bitwise complement. All of the bitwise operators yield **None** only when one of their operands is **None**.

### 16.3.4 String Operators

The following operators are available for operating on strings:

Operator	Description	Example
<b>+</b>	String concatenation	"Hello " + "World"
<b>+=</b>	Concatenation assignment	<b>\$result</b> += "here"
<b>~~</b>	Regex Matching	<b>\$s</b> ~~ "^d+"
<b>in</b>	Substring membership	"X" in <b>\$s</b>
<b>str</b>	String conversion	<b>str</b> ("X is " X)

When the operands to the **+** operator are both Strings, the result is a String value that represents the concatenation of the LHS and RHS operands. String variables may be appended to using the **+=** operator with the variable name appearing on the LHS and a String expression on the RHS.

The **~~** operator is a regular expression pattern matching and extraction operator. The LHS operand is a String expression and the RHS is a string literal that represents a PCRE regular expression. The result of the expression is the first substring in the LHS that matches the provided regular expression or **None** if there is no match. Regular expression pattern errors are diagnosed when the query is parsed. See [Section 16.6 Regular Expression Basics](#) for details on the regular expression syntax supported by the **~~** operator.

The **in** operator has Boolean type and evaluates to **true** if the LHS String value appears in the RHS string and **false** otherwise.

The **str** operator takes a parenthesized argument list consisting of one or more arbitrary expressions of non-void type. Each expression is evaluated and its result is converted to a textual representation based on its type as described in the table below.

#### 16.3.4.1 Textual Portrayal of Types



Type	Portrayal Description	Portrayal Example
ASTNode	If the node value can be converted to a <code>NamedDecl</code> , the fully qualified name of the <code>NamedDecl</code> . Otherwise, if the node is not <code>None</code> , the value of the builtin function <code>getDeclKindName</code> or <code>getStmtClassName</code> enclosed in angle brackets. If the node does not hold a value, the name of the static type of the node enclosed in angle brackets.	<code>B::foo</code>
ASTType	The textual value of the string type as per the builtin <code>getAsString</code> function.	<code>int *</code>
Boolean	<code>true</code> or <code>false</code>	<code>true</code>
Floating	Decimal representation corresponding to the C library's <code>%f</code> <code>printf</code> specifier.	<code>123.456</code>
Integral	Decimal representation with negative numbers prefixed with a minus sign.	<code>-123</code>
Location	<code>&lt;Invalid Location&gt;</code> , <code>&lt;Unknown Location&gt;</code> , or the values of the location buffer name, line number, and column number, separated by colons and enclosed in angle brackets.	<code>&lt;test.c:829:25&gt;</code>

The value of String expression arguments are unchanged. Expressions with a `None` value are represented as `<None>`.

### 16.3.5 The AST

Most queries will access the AST that PC-lint Plus uses to represent a source module. This AST consists of over 100 different kinds of nodes hierarchically related as shown in the diagram on the next page. Queries provides the `ASTNode` type, which may hold any kind of node used in the AST, as well as derived types corresponding to each node kind which may hold the specified node kind or any node kind derived from it. For example a `FunctionDecl` type may hold `CXXMethodDecl` nodes but not `VarDecl` nodes since `VarDecl` is not derived from `FunctionDecl`.

#### 16.3.5.1 Conversion Functions

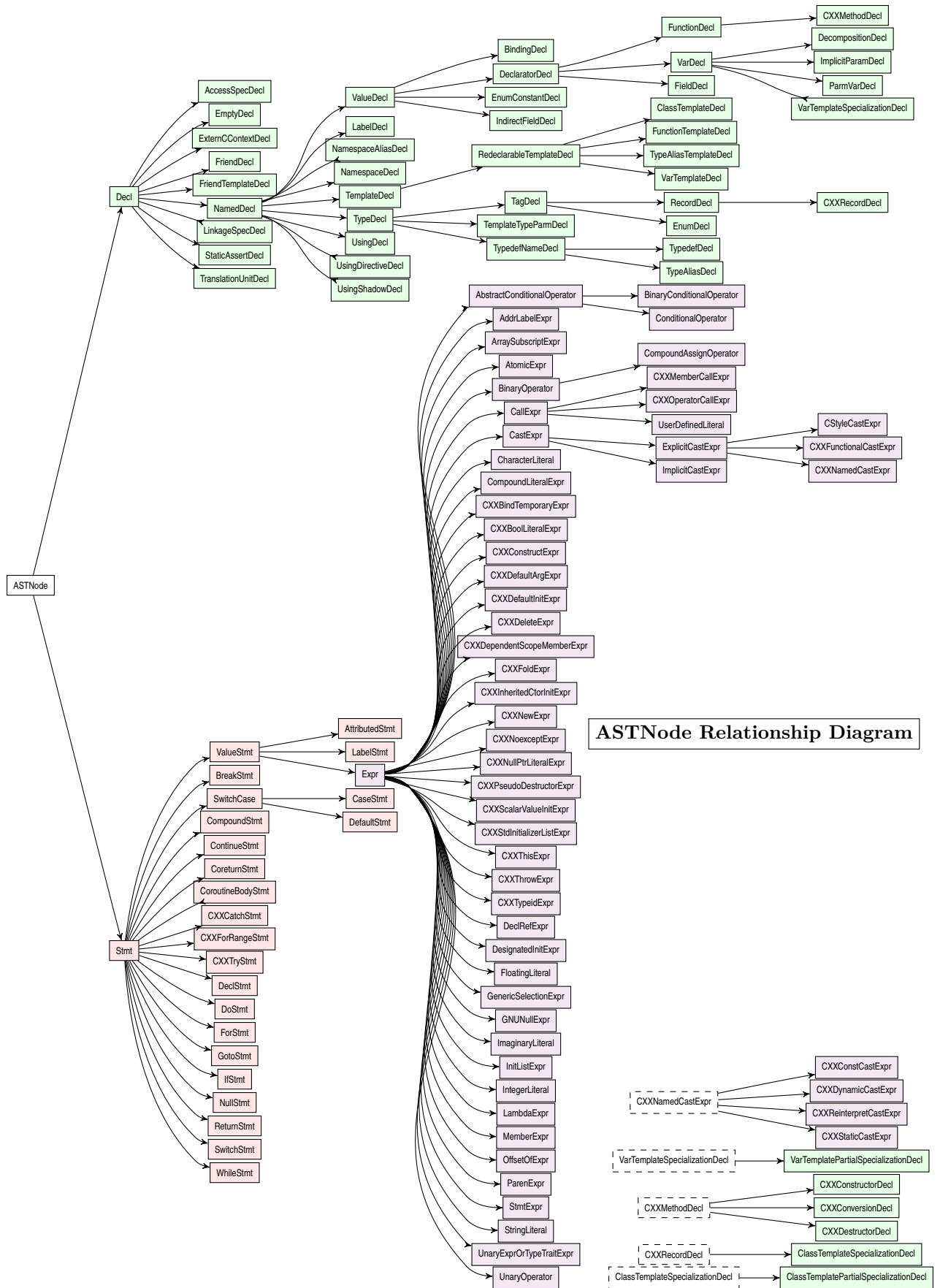
For each node type there is a conversion function with the same name that will convert a target node to the specified node type if the target node has the specified type or a type derived from it. If the conversion is successful, the result has the specified node type, otherwise the result is `None`. Conversion functions are used both to query the node type as well as to call node-specific functions available for individual node types. Conversion functions operate on `currentnode` if no argument is provided. For example, the query:

```
ContinueStmt()
```

will match any nodes that represent a `continue` statement. Within a `-astquery` option, the `currentnode` starts out with a type of `ASTNode`, to invoke functions that operate on a different node type, a conversion function must be called to convert the node to the appropriate type. For example, the builtin function `getNumParams` may only be called on a node with `FunctionDecl` type (or a derived node such as `CXXMethodDecl`, or `CXXConstructorDecl`). The following query will match functions declared with more than three parameters:

```
FunctionDecl().getNumParams() > 3
```





### 16.3.5.2 The `currentnode` Operator

During query evaluation there is the concept of a *current node* which by default corresponds to the AST node currently being visited for `-astquery` options or the anchored statement or expression of a message (if any) for `-equery/+equery` options (the `+paraminfo` option may be used to determine the statement or expression a message is anchored to). The `currentnode` operator yields the current node with a type of `ASTNode`. The *with-node* operator (described below) can be used to change both the value and static type of the `currentnode` operator. Because `currentnode` is used as the implicit argument for functions that accepts a compatible type as their first argument, it typically is not used explicitly.

### 16.3.5.3 The *with-node* (`:`) Operator

The *with-node* operator allows an expression to be evaluated in the context of a different `ASTNode`. The syntax for a *with-node* expression is:

```
node-expression : { body }
```

where *node-expression* is an expression with `ASTNode` type and *body* is one or more arbitrary expressions.

The *with-node* operator evaluates *body* with a possibly-modified *current node* value and/or type and yields the resulting value of *body* (or `None` if *node-expression* was `None`). For example, to issue a message when a static function definition is encountered which returns `void` and has no parameters, one could use:

```
FunctionDecl
FunctionDecl.isThisDeclarationADefinition
FunctionDecl.isStatic
FunctionDecl.getReturnType.isVoidType
FunctionDecl.getNumParams == 0
message(8001 "static function returns void but takes no arguments")
```

The calls to the `FunctionDecl` conversion operator are necessary because `isStatic`, etc. can only be called with a `FunctionDecl` object but implicit current node object has `ASTNode` type. The query can be rewritten using a *with-node* expression:

```
FunctionDecl : {
    isThisDeclarationADefinition
    isStatic
    getReturnType.isVoidType
    getNumParams == 0
    message(8001 "static function returns void but takes no arguments")
}
```

which produces the same behavior but removes the repeated use of `FunctionDecl`. If the current node is not a function declaration then `FunctionDecl` will be `None` and the body of the *with-node* expression will not be evaluated. Otherwise, the body will be evaluated in a context where the *current node* is a `FunctionDecl` instead of an `ASTNode` and functions expecting a `FunctionDecl` argument can be called using the implicit default `currentnode` argument.

In the above example, the *value* of the *current node* did not change, only its type. The value of the *current node* can also be modified using the *with-node* operator if the *node-expression* refers to a different AST node.

### 16.3.6 Relational Operators

The operands to the `<`, `>`, `<=`, and `>=` binary infix relational operators must both be `Numeric`, `String`, or `Location` types. If one operand is `Integral` and the other is `Floating`, the `Integral` operand is first converted to a `Floating` type. If the specified relation is true, the result is the value of the LHS operand, otherwise the result is `None`. If either operand is `None`, so is the result. For example the result of:

```
2 < 3
```

is 2 and the result of:

```
2 > 3
```

is **None**. Relational operators are right associative, this means that:

```
2 < $x < 5
```

yields a non-**None** value if and only if **\$x** has a value of 3 or 4. If either operand is empty, so is the result. The operands of the **==** operator must both have the same non-void expression types. The result of **==** has the same type as its operands and the value is the value of evaluating the LHS operand if both operands have the same non-**None** value and **None** otherwise.

The operands of the **!=** operator must both have the same non-void expression types. The result of **!=** has Boolean type and will never yield an empty value. Given expressions A and B, the result of the expression **A != B** is semantically equivalent to **not (A == B)**. In particular, if either A or B are empty, the result of the **!=** operator will be true regardless of the value of the other operand.

### 16.3.7 Variables and Assignment

A variable has a static type that is deduced from the first use of the variable which must be an assignment. Variable names may consist of letters, numbers, underscores, and dollar signs (\$) and must begin with a \$. Variable names are case sensitive and all characters in a variable's name are significant. Variables declared in one query are not accessible from another query.

All of the assignment operators have Void type, a side effect being that chained assignment is not supported and the value of assignment may not be used. All assignment operators take a variable name as a LHS argument and an expression of the same type on the RHS. In all forms of assignment, the RHS expression is first evaluated. For simple assignment (**=**), the result is assigned to the variable, if the RHS evaluates to an empty value, the variable now has an empty value.

Conditional assignment (**=?**) assigns the result to the LHS variable only if the RHS is not **None**. Compound assignment operates in the traditional manner, with **A x= B** having behavior similar to **A = A x B** where **x** is one of **\***, **/**, **%**, **+**, or **-** with the following exception:

If B is **None**, **A = A x B** will result in A having a value of **None** whereas **A x= B** will not (the value of A will not be changed).

The values of regular variables are reset at the beginning of every evaluation of a query. A *persistent* variable is one whose value persists across evaluations of a query, a persistent variable may be declared by preceding the initial assignment of the variable with the **persistent** keyword, e.g.:

```
persistent $counter = 0
```

The initial assignment to a persistent variable only occurs one time. Each analyzed module has its own set of persistent variables. Persistent variables are typically used as per-module counters, often with the **echo** operator.

### 16.3.8 The message Operator

The **message** operator is used to emit custom messages during the evaluation of an *AST query* and has the syntax:

```
message( [location-expression] message-number expr ... )
```

The optional *location-expression* is an arbitrary expression with Location type, the value of which will determine the file, line, and column number provided in the message as well as the context line and location of the position indicator. If *location-expression* is omitted, the location used defaults to the location associated

with the current node. If the resulting location has an empty value, the message is issued without location information.

*message-number* is a integer literal between 8001 and 8999, this is the message number that will be used to issue the message.

Following the message number is one or more expressions of arbitrary type which are evaluated and used to form the text of the emitted message. The result of each expression is converted to a string as described by the **str** operator and the message text is the concatenation of these strings. Expressions with a **false** or **None** value are converted to their textual representations and emitted with the message. The **message** operator always yields **true**.

Each expression will also be represented as a message parameter allowing suppression with the **+estring/-estring**, **+esym/-esym**, and **+etype/-etype** options. Expressions of **NamedDecl** and **ASTType** type will correspond to *symbol* and *type* parameters, respectively, all other expressions will correspond to *string* parameters. The **+paraminfo** option may be used to see the individual message parameters and types. Note that AST node expressions must have **NamedDecl** type, attempting to use an expression with another AST node type will result in a parse error (use the **str** operator to convert such nodes to a String value first). A maximum of nine expressions may be used with the **message** operator.

An argument to the **message** operator that is a compound expression containing exactly two sub-expressions of Location type is treated as a *location range*, the corresponding range will be highlighted in the line containing the position indicator if the range occurs on the same line as the emitted context line. *Location range* arguments to not contribute to the limit of nine expressions in a **message** operator. Except for location range arguments, an expression argument may not have Location type.

### 16.3.9 The **verbosify** Operator

The **verbosify** operator takes a single String operand and prints the value of the operand to the PC-lint Plus verbosity stream (**stdout** by default). The syntax for the **verbosify** operator is:

```
verbosify( string-expr )
```

**verbosify** yields a value of **true** unless its operand is **None** in which case nothing is printed and the result of the operator is **None**. Multiple strings may be combined into a single argument using the **+** or **str** operators and non-String expressions may be converted to a string using **str**.

Note that query verbosity messages are not shown by default, the verbosity option **-vq** can be used to cause verbosity messages emanating from a query to be displayed.

### 16.3.10 Logical Operators

The logical operators are **&&**, **||**, and **!**. **&&** and **||** are binary infix operators which accept any operands of non-Void type. The **&&** and **||** operators have Boolean type. The **&&** operator yields **true** when both of its operands have *truthy* values and **false** otherwise. The **||** operator yields **true** when either of its operands have *truthy* value and **false** otherwise. **&&** and **||** employ short-circuited operation, if the value of the expression can be determine from the evaluation of the LHS operand then the RHS operand will not be evaluated.

**!** is a prefix operator that accepts any non-Void operand and yields a Boolean value. If the operand has a *truthy* value, the **!** operator yields, **false**, otherwise it yields **true**.

None of the logical operators ever yield **None**.

#### 16.3.10.1 Alternate Spellings

The **&&**, **||**, and **!** operators may also be spelled **and**, **or**, and **not**, respectively.

### 16.3.11 The if Expression

An **if** expression has the form:

```
if condition { then-expr } [ else { else-expr } ]
```

The *condition* is first evaluated, followed by *then-expr* or *else-expr* (if provided) if the result of evaluating *condition* has a *truthy* or *falsy* value, respectively.

#### 16.3.11.1 The Type and Value of an if Expression

When an **if** expression contains an **else** clause and the type of *then-expr* and *else-expr* are of compatible types, the result of the **if** expression has the composite type of *then-expr* and *else-expr* and the value of whichever of the two expressions were evaluated. For example, the **if** expression:

```
if $x == 1 { 2 } else { 3.0 }
```

will have Floating type (the composite type of Integer and Floating) and will yield the value 2.0 if *\$x* is 1 and 3.0 otherwise.

When an **if** expression contains an *else-expr* which is not compatible with the *then-expr*, the type of the **if** expression is Boolean and yields **true** if the evaluated clause yields a *truthy* value and **false** otherwise. For example, the **if** expression:

```
if $x == 1 { FunctionDecl } else { CXXRecordDecl }
```

will yield **true** if *\$x* is 1 and the current node is a *FunctionDecl* or *\$x* is not 1 and the current node is a *CXXRecordDecl*, otherwise the **if** expression yields **false**.

When an **if** expression does not contain an **else** clause, the type of the **if** expression is Boolean and the result is **false** if the *condition* evaluates to a *truthy* value and *then-expr* evaluates to a *falsy* value, otherwise the result of the **if** expression is **true**. For example:

```
if 1 == 1 { 1 == 2 }
```

yields **false** but:

```
if 1 == 2 { ... }
```

will always yield **true**.

### 16.3.12 The while Expression

A **while** expression has the form:

```
while condition { do-expr }
```

The *condition* is first evaluated, if the result is a *truthy* value then *do-expr* is evaluated. This process repeats until *condition* evaluates to a *falsy* value at which point evaluation continues with the next expression. A **while** expression has Void type.

### 16.3.13 The echo Operator

AST queries are typically executed one time for each node visited during an AST pre-order traversal. It is sometimes desired to perform a second post-order visit to make use of information collected during the traversal of child nodes. For example, to report when the nesting depth of **if** statements within a function exceeds three, a nesting depth counter needs to be incremented during pre-order visitation of an **if** statement but also needs to be decremented during post-order visitation of **if** statements. The **echo** operator provides a mechanism by which select components of a query may be conditionally evaluated in post-order fashion.

The syntax for the `echo` operator is:

```
echo { body }
```

When an `echo` expression is evaluated, its body is *not* evaluated but instead is stored as a *reflection expression* that will be evaluated after the current node's children are traversed; the current node is then *echoed* and any *reflection expressions* associated with the echo are evaluated within the context of the echoed node. For example, the query:

```
persistent $previous_depth = 0
persistent $depth = 0

if FunctionDecl {
    $previous_depth = $depth
    $depth = 0
    echo { $depth = $previous_depth }
}

if IfStmt {
    $depth += 1
    echo { $depth -= 1 }
    if $depth > 3 {
        message(8001 "inappropriate nesting level for 'if'")
    }
}
```

will report when the `if` nesting depth within a function exceeds three. The persistent variable `$depth` is initialized to 0 (persistent assignments only occur once and persistent variables retain their values across query executions). When an `if` statement is encountered, `$depth` is incremented and the `echo` expression (but not its body) is evaluated which causes the body to be registered as a reflection to be executed when the current node is echoed. If `$depth` is now greater than 3, a message is issued. After any children are visited, the current `if` node is echoed and `$depth` is decremented. When a function declaration is encountered, the current depth is saved and then restored at the end of traversing the declaration using another `echo` expression. This is done so that `if` statements in nested functions do not contribute to the nesting depth of their enclosing function, e.g. no message is issued for:

```
void foo() {
    if (1) { if (2) { if (3) {
        []() { if (1) { if (2) { } } } };
    }}}
}
```

since the lambda starts with a new nesting depth of zero (if another nested `if` appeared immediately after the lambda expression, this would be reported as expected).

The body of the `echo` expression forms a scope in which non-persistent variables defined outside the expression are represented by local copies starting with the captured values held at the time the `echo` expression was evaluated to register the echo, and non-persistent variables defined in the body cannot be accessed outside the body. Echoes are registered in the order in which they are encountered during evaluation and the reflections are evaluated in the same order. For example, to report all instances of a typedef defined within a non-inline namespace in:

```
namespace X {
    typedef int A;
    namespace Y {
        typedef int B;
        namespace Z {
```

```

        typedef int C;
        inline namespace W {
            typedef int D;
            namespace P {
                typedef int E;
            }
        }
    }
}

```

a persistent variable storing the current namespace can be maintained during the evaluation of `NamespaceDecl` nodes (using an `echo` to restore its previous value potentially referring to an enclosing namespace from a captured value) and this current namespace variable can be accessed during the evaluation of a `TypedefNameDecl` to check if the current namespace is inline:

```

persistent $current_namespace = NamespaceDecl
if NamespaceDecl {
    NamespaceDecl : {
        $pending_namespace = $current_namespace
        echo { $current_namespace = $pending_namespace }
        $current_namespace = currentnode
    }
} else if TypedefNameDecl {
    TypedefNameDecl : {
        if not $current_namespace.isInline {
            message(8001
                "typedef '" currentnode
                "' defined in non-inline namespace '"
                $current_namespace "'"
            )
        }
    }
}

```

which will report typedefs A, B, C, and E but will not report typedef D because namespace W is inline.

Note that since the body of an `echo` expression is not evaluated until a node is echoed, and nodes are never echoed more than once, a nested `echo` operator would have no effect as the body of the inner `echo` would only be registered during the `echo` and never have the opportunity to be reflected. For this reason, a query that contains a nested `echo` will produce a parse error.

#### 16.3.14 Compound Expressions

A compound expression consists of one or more expressions enclosed by braces (`{}`). The type of the compound expression is the type of the last enclosed expression. Expressions appearing within a compound expression are evaluated sequentially. If the evaluation of any of these expressions yields a value of `false` or `None`, the remaining expressions are not evaluated and the value of the compound expression is `None`. Otherwise, the value of the compound expression is the value of the last expression. For example:

```
{ 1 2.0 }
```

consists of two expressions (1 and 2.0) with types `Integral` and `Floating`, respectively. The type of the compound expression is `Floating` and the value will be 2.0 when evaluated. The compound expression:

```
{ false 1 }
```

has type `Integral` but will always yield a `None` value during evaluation as the expression 1 will never be reached since evaluation of the compound expression will stop after evaluating the expression `false`.

### 16.3.15 Parentheses

Parentheses may be used as a *grouping operator* to enclose any expression (note that e.g. the LHS of assignment expects a variable name, not an expression, so `$x = 1` is valid but `($x) = 1` is not). Parentheses must enclose exactly one expression, e.g. `()` or `(1 2)` are syntax errors; braces can be used to enclose multiple expressions. The primary purpose of parentheses is to override operator precedence. There is never any performance impact for using parentheses.

Braces may also be used to override precedence and `{X}` is functionally equivalent to `(X)` for some expression `X` (recall that the type and value associated with a compound expression is that of its last enclosed expression). There is no performance impact for using braces to override operator precedence in this way.

Parentheses are required to enclose the explicit argument list in a function invocation and are required to enclose the argument(s) of function-like operators including `assert`, `message`, and `str` although the parentheses are not technically operators in these cases.

### 16.3.16 Operator Precedence

The following table lists the precedence of query operators. Operators with a smaller precedence value bind more tightly than those with a higher value. All operators have left-to-right associativity except the with-node `(:)` operator and the relational operators which are right-to-left associative. Parentheses or braces may be used to modify expression precedence, e.g. `1 + 2 * 3`  $\equiv$  7 but `(1 + 2) * 3`  $\equiv$  9.

Precedence	Operator(s)	Description
1	<code>()</code>	Parentheses
2	<code>.</code>	Member Call
3	<code>-</code>	Unary Negation
4	<code>~</code>	Bitwise Complement
5	<code>&amp;</code>	Bitwise And
6	<code> </code>	Bitwise Or
7	<code>* / %</code>	Multiplicative Operators
8	<code>+ -</code>	Additive Operators
9	<code>!= == &lt;= &gt;= &lt; &gt;</code>	Relational Operators
10	<code>in</code>	In Operator
11	<code>!/not</code>	Logical Not
12	<code>&amp;&amp;/and</code>	Logical And
13	<code>  /or</code>	Logical Or
14	<code>= =? *= /= %= += -=</code>	Assignment

The relational operators are right-to-left associative, all other operators are left-to-right associative.

## 16.4 Builtin Functions

Much of the functionality provided by queries is realized through the use of hundreds of builtin functions. A builtin function accepts zero or more arguments of predefined types and returns a value of the type associated with the function. The expected number and type of arguments is checked at parse time. Function arguments are evaluated from left to right. If any of the provided arguments have an empty value at run-time, the remaining arguments are not evaluated, the function is not called, and the result of the function is an empty value.

### 16.4.1 Function Call Syntax

A builtin function is invoked by writing the name of the function followed by an optional parenthesized argument list, e.g.:



```
isStruct(currentnode)
```

If there are no trailing arguments, an empty parenthesized list is allowed but optional.

The first argument of any function may be specified by placing it on the left-hand side of the `.` member call operator, e.g.:

```
currentnode.isStruct()
```

As mentioned above, the parentheses here are optional. The member call syntax provides a convenient, and perhaps more intuitive, syntax when using nested function invocations. For example, the query:

```
isStruct(getParent(FieldDecl(currentnode)))
```

used to determine if the current node is a `FieldDecl` whose parent is a `struct` can be written as:

```
currentnode.FieldDecl.getParent.isStruct
```

There is no functional difference between the syntaxes.

### 16.4.2 Default Arguments

Most functions accept an `ASTNode` type as their first (and typically only) argument. In such cases, the argument will default to the type and value of `currentnode` if the function is called with one argument less than the number of parameters expected by the function. For example, all of the `ASTNode` conversion functions take a single `Node` argument and thus:

```
IsStmt ≡ IsStmt(currentnode)
```

Similarly:

```
isDerivedFrom($class) ≡ isDerivedFrom(currentnode, $class) ≡ $class.isDerivedFrom
```

Note that the type of the `currentnode` operator is context-dependent. When appearing in the `-astquery` option, it starts with a type of `Node` but its type may change within a `with-node` (`:`) operator. If the type of the `currentnode` operator is not compatible with the type expected as the first argument, an error will be produced when the query is parsed. In the above example, since `isDerivedFrom` expects two `CXXRecordDecl` arguments, the implicit use of `currentnode` will result in an error outside of a context where the current node has `CXXRecordDecl` type or type derived therefrom.

### 16.4.3 Naming Conventions

Builtin functions typically start with a lowercase letter and employ camelCase having names that match the corresponding underlying clang functions that they typically wrap. All of the `ASTNode` conversion functions (e.g. `IfStmt`, `RecordDecl`, etc.) start with an uppercase letter and match the name of the corresponding clang class. This facilitates easy transition between the clang-produced program AST and query syntax and helps conversion functions stand out from other functions.

## 16.5 Examples

### 16.5.1 -astquery Examples

This section contains over a dozen examples of how the `-astquery` option may be used to craft custom messages, many of which correspond to real-world uses cases provided by customers. Each example begins with a short problem description followed by a query that implements a corresponding solution and elaborative notes about the provided query.

**Non-local variable names should be at least 3 characters long**

```
-astquery(VarDecl() : {
    !isLocalVarDeclOrParm() && getNameAsString().length() < 3
    message(8001 "variable name '" currentnode "' is too short")
})
```

Variable declarations are represented by `VarDecl` nodes in the AST, the `VarDecl()` conversion function will only match such nodes. The `isLocalVarDeclOrParm` function is used to exclude function parameters or variables declared with function scope. The `getNameAsString` function returns a string containing the name of the variable and the `length` function returns the length of this string. The message will be issued with the name of the variable as a *symbol* parameter which may be suppressed using the `-esym` option.

### Variables of unsigned integer type should have a name starting with "uint"

```
-astquery(VarDecl() : {
    getType().isUnsignedIntegerType()
    hasName() && !getNameAsString().startsWith("uint")
    message(8001 "unsigned variable '" currentnode "' missing 'uint' prefix")
})
```

The `getType` function returns the type of a declaration (as an `ASTType`), in this case the type of the current variable declaration. The `isUnsignedIntegerType` function is called with an argument of `ASTType` and returns `true` if the type is an unsigned integer type. The `getNameAsString` function is used to obtain the variable name and the `startsWith` function returns `true` if the first argument (the result of calling `getNameAsString`) starts with the value provided in the second argument ("uint"). The `hasName` function is used to ensure the variable has a name so as to not emit a message for e.g. a function declaration with unnamed parameters. The message reports the offending variable as a *symbol* parameter.

### Constructors and destructors should not be inline

```
-astquery(FunctionDecl() : {
    CXXConstructorDecl() || CXXDestructorDecl()
    !isDefaulted() && !isDeleted()
    isInlined()
    message(8001 "ctors and dtors should not be inline")
})
```

Constructors and destructors are represented by `CXXConstructorDecl` and `CXXDestructorDecl` nodes, respectively. Both of these are derived from `FunctionDecl` which is used with the *with-node* operator so that the remaining function calls, which take a `FunctionDecl` argument, may be called with the implicit `currentnode` object. The `isDefaulted` and `isDeleted` functions return `true` for defaulted (= default) and deleted (= delete) functions, respectively. This check is included to prevent the message from being issued for deleted or defaulted functions as these would be considered to be inline functions which is likely not desired for the purpose of this message. The `isInlined` function returns `true` if the function is inline.

### Bit-field should not be defined

```
-astquery(FieldDecl() : {
    isBitField()
    message(8001 "bit-field defined")
})
```

Non-static data-members of aggregates are represented as `FieldDecl` nodes in the AST. The `isBitField` function takes a `FieldDecl` argument (the `currentnode` object is implicitly supplied as an argument in this case which has `FieldDecl` type by virtue of the *with-node* operator) and returns `true` if the specified field is a bit-field.

### Variable Length Arrays should not be used

```

-astquery(VarDecl() : {
    getType().isVariablyModifiedType()
    message(8001 "variable '" currentnode "' has VLA type")
})

```

The `isVariablyModifiedType` function is used to detect and report variable-length array declarations. Instances of the message are parameterized with a *symbol* representing the offending variable.

#### Variables of function pointer type should be declared using typedefs

```

-astquery(VarDecl() : {
    getType().isFunctionPointerType()
    !getType().isTypedefNameType()
    message(8001 "use typedef to declare function pointer variables")
})

```

When a variable is declared using a typedef, the `getType` function will return the typedef type for which the function `isTypedefNameType` will return true. The function `isFunctionPointerType` takes an `ASTType` and returns true if the type is a function pointer type (regardless of whether it is a typedef type).

#### Non-public, non-struct class members should end with a trailing \_

```

-astquery(FieldDecl() : {
    getAccess() != "public"
    getParent().isClass()
    hasName() && !getNameAsString().endsWith("_")
    message(8001 "non-public class member '" currentnode "' should end with _")
})

```

The `getAccess` function returns a string that indicates the access of a declaration (public, private, protected, or none). The `getParent` function returns the `RecordDecl` that contains the provided `FieldDecl` and the `isClass` function determines if the that `RecordDecl` was declared with the `class` keyword. If the field does not have public access, the parent has class type, and the name of the field does not end with `_` then a message is issued with the field provided as a *symbol* parameter.

#### Constructors that are callable with a single argument should be made explicit except for copy/move constructors and constructors taking only a `std::initializer_list` argument.

```

-astquery(CXXConstructorDecl() : {
    getMinRequiredArguments() == 1
    !isCopyOrMoveConstructor()
    !isExplicit()
    !getParamDecl(1).getType().isStdInitList()
    message(8001 "single arg non-copy/move ctors must be marked explicit")
})

```

The `getMinRequiredArguments` function returns the number of arguments that must be provided when calling a function, `isCopyOrMoveConstructor` returns true if the provided function is a copy or move constructor, and `isExplicit` returns true if the function was declared with the `explicit` keyword. The `getParamDecl` function returns the specified parameter, `getType` is used to obtain the type of the first parameter, and `isStdInitList` is used to determine if the type of this parameter is an instantiation of `std::initializer_list`. If the constructor requires exactly one argument, is not a copy or move constructor, is not declared with the `explicit` keyword, and does not accept a `std::initializer_list` argument, a custom message will be emitted.

#### Default function arguments should be avoided except in constructors and static functions defined outside a header

```

-astquery(ParmVarDecl() : {
    hasDefaultArg()
    getParentFunctionOrMethod() : {
        !CXXConstructorDecl()
        !(getStorageClass() == "Static" && getDefinition().getLocation().isMainFile())
    }
    message(8001 "default arg not allowed for parameter '" currentnode "'")
})

```

The above query will be executed for every parameter variable. The `hasDefaultArg` returns true if the parameter has a default argument and `getParentFunctionOrMethod` is used to obtain the function that the parameter belongs to. If the parent function is not a constructor (determined by calling `CXXConstructorDecl` and is not a static function defined in the main source file, a message is emitted with a *\*symbol\** parameter representing the parameter variable. The `getDefinition` function is used to obtain the function declaration that contains the body of the function and the location of this function is checked to determine if it resides in the main source file via `isMainFile`.

**The construct for (;;) ... should not be used to represent an infinite loop**

```

-astquery(ForStmt() : {
    !getInit() && !getCond() && !getInc()
    message(8001 "infinite loop via 'for (;;)'" )
})

```

The `getInit`, `getCond`, and `getInc` functions return the first, second, and third clauses of a `for` statement, respectively. Each of these functions will return a value of `None` if the corresponding clause is not provided. If all of these functions return `None` then the `for` statement has the proscribed form.

**Enumeration constants should consist entirely of uppercase letters, digits, and underscores**

```

-astquery(EnumConstantDecl() : {
    !getNameAsString() ~~ "[A-Z_0-9]+$"
    message(8001 "enumeration constant '" currentnode
        "' should contain only uppercase letters, digits, and underscores")
})

```

Enumeration constants are represented by `EnumConstantDecl` nodes in the AST so the conversion function of the same name is used to only match those nodes. The `getNameAsString` function is used to get the constant's name and the `~~` pattern matching operator is used to check whether the name consists entirely of the allowed characters.

**Structures (declared with the `struct` keyword) should have names ending with `__t`**

```

-astquery(RecordDecl() : {
    isStruct() && hasName() && !getNameAsString().endsWith("__t")
    message(8001 "structure '" currentnode "' does not contain '_t' suffix")
})

```

Structures, classes, and unions are all represented by the `RecordDecl` AST node, the `isStruct` function is used to determine if this record was declared with the `struct` keyword. If it was, the `getNameAsString` function is used to obtain the name and the `endsWith` function checks to see that the name contains the requisite suffix.

**Bit-fields should not be defined using a type smaller than `int`**

```

-astquery(FieldDecl : {
    isBitField() && getType().isPromotableIntegerType()
    message(8001 "type '" getType() "' not allowed for bit-fields")
})

```

Similar to a previous example, the `FieldDecl` and `isBitField` functions are used to isolate bit-fields. The `getType` function, applied to a bit-field, will yield the type used in the declaration of the bit-field. The `isPromotableIntegerType` function returns true if the provided type is an integer type that is eligible for promotion which is only the case for integer types smaller than `int`. The type used in the declaration of the offending bit-field is included as a *type* parameter in the message which may be used to suppress the message with the `-etype` option.

**The `strncpy` function should not be called with the same variable for the first two arguments**

```
-astquery(CallExpr() : {
    getDirectCallee().getNameAsString() == "strncpy"
    getArg(1).ignoreParenCasts().DeclRefExpr().getDecl().VarDecl().getNameAsString() ==
    getArg(2).ignoreParenCasts().DeclRefExpr().getDecl().VarDecl().getNameAsString()
    message(8001 "source and destination argument for strncpy are the same")
})
```

The `getDirectCallee` function returns the `FunctionDecl` corresponding to the `CallExpr` or `None` if the function cannot be determined (e.g. if the function was called through a function pointer). The name of the called function is then checked and the query will only continue for calls to `strncpy`. The `getArg` function is used to access the first and second arguments of the call, `ignoreParenCasts` will skip over any parentheses and casts and `DeclRefExpr` will return AST node of the same name if such a node is present. A `DeclRefExpr` node is used to represent a reference to a declared entity. For example, in the expression `a + b`, the references to the variables `a` and `b` will each be represented by a `DeclRefExpr` in the AST. The `getDecl` retrieves the declaration corresponding to the `DeclRefExpr` and the `VarDecl` function will yield the declaration as a `VarDecl` if it is a variable declaration. The `getNameAsString` function is then used to see if both arguments name the same variable.

**A C-style cast should not be used to convert between pointer types**

```
-astquery(CStyleCastExpr() : {
    $source_type = getSubExpr().ignoreParenImpCasts().getType().getNonReferenceType()
    $dest_type = getType().getNonReferenceType()
    $source_type.isPointerType() && $dest_type.isPointerType()
    message(8013 "c-style cast used to convert from '" $source_type
        "' to '" $dest_type "'")
})
```

The `CStyleCastExpr` represents a c-style cast (e.g. one of the form `(type) value`). When applied to a `CStyleCastExpr`, the `getType` function returns the type casted *to*. The `getSubExpr` function returns the expression being casted and calling `getType` on this expression yields the type being casted *from*. The `getNonReferenceType` function returns the type that results from removing any reference types so that e.g. a variable having reference-to-pointer type is still eligible to be reported by this message. If both types are pointer types (determined using the `isPointerType` function), the message is issued. The emitted message contains two *type* parameters which provide additional information about the cast and may be used to suppress the message for casts involving specific types.

### 16.5.2 -equery Examples

The below examples demonstrate various ways that the `-equery` option may be used to suppress messages in ways that cannot be suppressed using standard suppression options. Most of the examples are taken from real-world use cases. Each example begins with a description of the desired suppression and is followed by a query that implements the suppression along with relevant explanation.

**Suppress message 9033 (cannot cast essential-type to wider/different essential type) when casting to a typedef named `EXEMPT`**

```
-equery(9033, CastExpr().getType().getAsString() == "EXEMPT")
```

Message 9033 is issued for a cast expression which is accessible via the *current node*. The current node is converted to a `CastExpr` and the cast type is accessed via the `getType` function. If the type is a typedef with a name of `EXEMPT`, the message will be suppressed.

**Suppress message 9130 (bitwise operator applied to signed underlying type) for the '|' operator when the LHS is an integer constant expression**

```
-equery(9130, BinaryOperator() : {
    getOpcodeStr() == "|" && getLHS().isIntegerConstantExpr()
})
```

The `BinaryOperator` corresponding to the bitwise operator for which message 9130 is issued is available as the *current node*. The `getOpcodeStr` function returns a string representation of the binary operator and the `getLHS` function will return the left-hand operand. When the operator is `|` and the left-hand expression is an integer constant expression (as per the `isIntegerConstantExpr` function), the message is suppressed.

**Suppress message 916 (implicit pointer assignment conversion) for conversions to void \***

```
-equery(916, msgTypeParam(2).isVoidPointerType())
```

Message 916 is parameterized by two *type* parameters, the type being converted *from* is the first *type* parameter and the type being converted *to* is the second. Message *type* parameters may be accessed using the `msgTypeParam` function which takes an index as its argument and returns the corresponding *type* parameter as an `ASTType` value. The query retrieves the *to* type and suppresses the message if it is a void pointer using the `isVoidPointerType` function.

**Suppress message 1731 (public virtual function) for pure functions in classes without in-class initializers**

```
-equery(1731, msgSymbolParam(1).CXXMethodDecl() : {
    isPure() && !getParent().hasInClassInitializer()
})
```

Message 1731 is parameterized by a single *symbol* parameter corresponding to the function that is the target of this message. The `isPure` function returns true if the provided function is a pure virtual function (e.g. `void foo() = 0`). The `getParent` function returns the class containing the member function and the `hasInClassInitializer` function returns true if this class has any in-class initializers.

**Suppress message 1925 (symbol is a public data member) for members that have a non-POD class, structure, or union type**

```
-equery(1925, msgSymbolParam(1).FieldDecl() : {
    getType().getNonReferenceType().isRecordType()
    !getType().getNonReferenceType().isPODType()
})
```

Message 1925 is parameterized by the *symbol* corresponding to the public data member (a `FieldDecl`) referenced in the message. The `getType` function returns the type of this member and the `getNonReferenceType` returns the type resulting from removing any references from the type. The resulting non-reference type of the data member is then checked to determine if it is a class, structure, or union type by the `isRecordType` function and the `isPODType` determines if the type is a POD type.

**Suppress message 9150 (non-private data member within a non-POD structure) for non-POD data members or data members that are arrays of non-POD types**

```
-equery(9150, msgSymbolParam(1).FieldDecl() : {
    !getType().getNonReferenceType().isPODType() ||
    (getType().isArrayType() &&
     !getType().getArrayElementType().isPODType())
})
```

Message 9150 is parameterized by the *symbol* corresponding to the data member referenced in the message. The `getType` function returns the type of this member and the `getNonReferenceType` returns the type resulting from removing any references from the type. The `isArrayType` function returns true for array types and the `getArrayElementType` function returns the element type of the array. The `isPODType` function returns true if the resulting type is a POD type. If the data member is not a POD type or is an array of non-POD type the message is suppressed.

**Suppress message 1732 (new in constructor for which has no user-provided copy assignment operator) for classes directly derived from a class named NonCopyable**

```
-equery(1732, msgSymbolParam(1).CXXRecordDecl() : {
    $num_bases = getNumBases()
    $base_idx = 1
    $should_suppress = false
    while $base_idx <= $num_bases {
        if getBase($base_idx).getAsCXXRecordDecl().getNameAsString() == "NonCopyable" {
            $should_suppress = true
        }
        $base_idx += 1
    }
    $should_suppress
})
```

Message 1732 is parameterized by the *symbol* corresponding to the class referenced in the message. The `getBase` function returns the  $n^{\text{th}}$  base of a class as a *type*, the `getAsCXXRecordDecl` function is used to obtain the corresponding class declaration, and the `getNumBases` returns the number of direct bases a class has. The `while` loop is used with these functions to iterate over the bases of the class mentioned in the message. If the name of any of these base classes (available using the `getNameAsString` function) is `NonCopyable` then the `$should_suppress` variable is set to `true` which prevents issuance of the message at the end of the query.

**Suppress message 953 (local variable could be const) for pointers**

```
-equery(953, msgSymbolParam(1).VarDecl().getType().isPointerType())
```

Message 953 is parameterized by the *symbol* corresponding to the variable referenced by the message and is accessed by the `msgSymbolParam` function. The `getType` function returns this variables type and the `isPointerType` returns true if this type is a pointer type.

**Suppress message 818 (parameter of function could be pointer to const) for functions instantiated from a template**

```
-equery(818, msgSymbolParam(2).FunctionDecl().isTemplateInstantiation())
```

Message 818 is parameterized by two *symbols*, the function parameter and the function itself. The `msgSymbolParam` function is used to access the `FunctionDecl` represented by the second *symbol* parameter and the `isTemplateInstantiation` function is used to determine if the function was instantiated from a function template.

**Suppress message 523 (expression statement lacks side effects) when issued for a call to a virtual function**

```
-equery(523, CallExpr().getDirectCallee().CXXMethodDecl().isVirtual())
```

The `CallExpr` conversion operator is used to access the call expression for which message 523 is issued. The `getDirectCall` function returns a `FunctionDecl` corresponding to the called function. The `isVirtual` function takes the `CXXMethodDecl` (which is why the conversion function of the same name is used before `isVirtual`) and returns `true` if the function is a virtual function and `false` otherwise.

Suppress message 835 (zero given as argument to operator) when the indicated side is an enumeration constant

```
-equery(835, msgStringParam(1).endsWith("right") && msgStringParam(2) == "|"
      BinaryOperator().getRHS().ignoreParenCasts().DeclRefExpr().getDecl().EnumConstantDecl())

-equery(835, msgStringParam(1).endsWith("left") && msgStringParam(2) == "|"
      BinaryOperator().getLHS().ignoreParenCasts().DeclRefExpr().getDecl().EnumConstantDecl())
```

Message 835 is parameterized by three *string* parameters: the first parameter indicates whether the argument appears on the *left* or *right* side of the operator, the second parameter is a string that represents the operator, and the third parameter is the context for which the message is issued. The `msgStringParam` function takes an index argument and returns the corresponding *string* parameter of the message. The above query first isolates instances of the message with the specified *string* parameters. Message 835 is issued for a `BinaryOperator` node. The remaining functions are used to determine if the appropriate operand of this operator constitutes an enumeration constant.

Suppress message 529 (local variable declared in function not subsequently referenced) for volatile-qualified variables

```
-equery(529, msgSymbolParam(1).VarDecl().getType().isVolatileQualified())
```

Message 529 is parameterized by two *symbol* parameters: the local variable and the function in which it is declared. The `msgSymbolParam` function is used to retrieve the first symbol which is converted to a `VarDecl`. The `getType` function provides the type of the variable and the `isVolatileQualified` function returns true if the type is volatile-qualified.

## 16.6 Regular Expression Basics

A regular expression is a string of characters that represents a search pattern used to match sections of text. Regular expressions consist of regular characters ('a', '1', '\_', etc.) that have a literal meaning and metacharacters ('\*', '+', '|', etc.) that have special meanings. Regular characters match themselves and special characters and sequences are used to specify more complex behavior. PC-lint Plus supports Perl-style regular expressions inside of queries via the `~~` pattern matching operator. This section provides an overview of regular expression syntax and discusses some of the basic concepts involved. Most of the patterns used with PC-lint Plus will be relatively simple and an understanding of the basic concepts will suffice.

The purpose of this section is not to provide an exhaustive reference of regular expressions (for which numerous books have been written) but to introduce the basic concepts and provide a reference for some of the more advanced topics that may be useful in the context of Hooks. For additional information, please see:

- [Perl's regular expression documentation](#)
- [PCRE2 pattern documentation](#)
- [Mastering Regular Expressions by Jeffrey E. F. Friedl](#)

### 16.6.1 Regular Characters and Metacharacters

All characters appearing in a regular expression are either regular characters or metacharacters. Regular characters represent themselves, e.g. the pattern "abc" will match the literal text "abc". Metacharacters instead impart a special meaning to the pattern, for example, "ab\*c+" will match text that contains the letter 'a', immediately followed by zero or more 'b' characters, followed by one or more 'c' characters (e.g. "acc" and "acccb" will match but "abb" won't). The metacharacters used in regular expressions are \*, ?, +, {, }, (, ), [, ], ^, \$, |, \, and ..



### 16.6.2 Pattern Anchoring

By default, patterns are not anchored meaning that a match anywhere in the string is considered success. For example, the pattern `"a+"` will match the text `"cab"`. The anchor characters `^` and `$` can be used to anchor the pattern to the beginning or end of the subject string, respectively. For example, `"^a+"` will match any string that starts with one or more `'a'` characters and `"a+$"` will match any string that ends in one or more `'a'` characters. `'^'` and `'$'` can be used together to specify an exact match of the subject string, e.g. `"^a+$"` will match only a string consisting entirely of `'a'` characters.

### 16.6.3 Dot and Repetition

The dot character `'.'` represents any single character, e.g. `"c.b"` will match `"cab"`, `"cub"`, `"ccb"`, etc. The `'?'`, `'*'`, and `'+'` characters are used to modify the number of times a immediately preceding subpattern should be matched. `'?'` means that the subpattern can match zero or one time, `'*'` means zero or more times, and `'+'` means one or more times. For example, `"c.?b"` will match `"cab"` and `"cb"`, but not `"club"` whereas `"c.+b"` would match `"cab"` and `"club"` but not `"cb"` and `"c.*b"` would match all three.

A set of curly braces can also be used to specify repetition limits. If one number appears inside of the braces, that is the number of times the preceding subpattern must match to be successful. Alternatively, a min and max may be specified, separated by a comma. In this case, the first number specifies the minimum number of times the subpattern can match and the second number specifies the maximum. For example, `"ca{2,3}"` will match `"caab"` and `"caaab"` but not `"cab"`. If the second number is elided, this is taken to mean there is no maximum, e.g. `"fi1,ght"` will match `"fight"`, `"fiight"`, `"fiiiight"`, etc.

### 16.6.4 Alternation

The `|` character specifies pattern *alternation*, e.g. either the preceding or the following pattern may match. For example, the pattern `"blue|red|green"` will match a string that contains either `"blue"`, `"red"`, or `"green"`.

### 16.6.5 Subpatterns

Parts of a pattern surrounded by parenthesis are *subpatterns*. A subpattern can have its own modifiers attached to it. For example, `"q(ue)+"` will match `"que"` and `"queue"` (and `"queueue"` etc.) but will not match `"qu"`. Subpatterns can be arbitrarily complex.

### 16.6.6 Backslash

The `'\'` character can be used to introduce backslash escape sequences and to remove the meaning from metacharacters. For example `"*"` is not a valid pattern because the `*` quantifier must be applied to a pattern. To match a literal `*` the pattern `"\*"` can be used; the backslash removes the special meaning. Note that the meaning is only removed for the character that immediately follows, e.g. `"\**"` will match zero or more `*` characters. In addition to removing the meaning from metacharacters, the following escape sequences are supported:

Escape Sequence	Meaning
<code>\d</code>	Any decimal digit (i.e. 0-9)
<code>\D</code>	Any non-decimal character
<code>\h</code>	Any horizontal whitespace character
<code>\H</code>	Any character except horizontal whitespace
<code>\s</code>	Any space character
<code>\S</code>	Any non-space character
<code>\v</code>	Any vertical whitespace character

Escape Sequence	Meaning
\V	Any character except vertical whitespace
\w	Any "word" character (letter, digit, or underscore)
\W	Any character that is not a word character

In addition to escape sequences that match classes of characters, escape sequences can also represent *assertions*, conditions that must be true at the point in which they appear for the pattern to successfully match. The following assertions are supported:

Escape Sequence	Meaning
\b	Matches at a word boundary
\B	Matches anywhere except a word boundary
\A	Matches at the start of the subject string
\Z	Matches at the end of the subject string
\G	Matches at the first matching position of the subject string

A word boundary is the start or end of a sequence of one or more word characters.

### 16.6.7 Character Classes

The '[' character starts a character class which ends with the corresponding ']' character. The characters in between the brackets are all part of the class and a match of any of them fulfills the corresponding portion of the pattern. This is useful for matching one of a set of characters. For example, "c[au]b" will match "cab" and "cub" but not "cb" or "cob" or "caub". The POSIX character classes listed below can also be used within a character class by enclosing the name in [: ... :], e.g. the pattern "[\_[:lower:][:digit:]]" will match a character that is either a lowercase letter, a digit, or the underscore. A character class cannot be empty and as such a ] can be included in a character class by specifying it as the first character of the class, e.g. [ ]abc].

Class Name	Description
alnum	letters and numbers
alpha	letters
ascii	ASCII characters
blank	space or tab
cntrl	control characters
digit	numbers (0-9)
graph	printing characters except space
lower	lower case letters
print	printing characters, including space
punct	printing characters that are not letters, digits, or space
space	whitespace
upper	upper case letters
word	word characters (letters, digits, and underscore)
xdigit	hexadecimal digits

Ranges may be used in a character class by using - between two characters, e.g. "[A-F0-9]" to match upper case hexadecimal digits. To include the - as part of the set, include it at the beginning or escape it with a backslash. Character class escape sequences can be used in bracketed character classes, e.g. "[a-f\d]" to match lower case hexadecimal digits.

A *negated* character class is one that matches anything *except* the types of characters included in the class. To negate a character class, use ^ as the first character in the class, e.g. "[^[:lower:]]\_" will match any character that is not a lower case letter or underscore character.

The metacharacters `$`, `|`, `(`, `)`, `?`, `*`, `+`, and `{` do not maintain their special meaning inside of a character class and can be used without escaping them. The `^`, `-`, and `[` characters are only considered special when appearing at the beginning of the pattern, in a location where a range operation is permitted, and when introducing a POSIX character class, respectively. In other parts of a character class, these characters have no special meaning.

### 16.6.8 Captures

A *capture* occurs any time a parenthesized subpattern is matched. It is called a *capture* because the text that matched the subpattern is saved and can be accessed later in the pattern. A reference to a capture within a pattern is called a *backreference* and can be referred to within the pattern by the capture number or capture name (if provided).

Capture groups are numbered starting with 1 for the first group, 2 for the second, etc. A backreference to a capture group with a number between 1-9 can be made using the syntax `\#`. For example the pattern `"ca(.)\1"` will match any string containing `"ca"` followed by two of the same letter, e.g. `"call"`. Backreferences can also be referred to using the syntax `\g{#}`, e.g. `"ca(.)\g{1}"`. The latter syntax avoids possible ambiguities between the syntax for specifying octal escape sequences as well as allowing the referencing of capture group numbers greater than 9.

Capture groups can be given names by starting the group with `?<name>` and named backreferences using the syntax `\k{name}`, e.g. `"ca(?<c1>.)\k{c1}"`. Named capture groups are useful in larger or more complex patterns making use of many capture groups and provide a form of documentation within the pattern.

### 16.6.9 Mode Modifiers

By default, matches are performed in a case-sensitive fashion. Case insensitive matches can be performed for part or all of a match by using the *i mode modifier*. A *mode modifier* is a syntax for temporarily changing the behavior of how patterns are interpreted. Mode modifiers are introduced by placing them between the `?` and `:` characters that start a non-capturing group. For example, the pattern `"(?i:foo)"` will match `"foo"`, `"FOO"`, `"Foo"`, etc. The case-insensitive behavior is applied to the containing group. An entire pattern, possibly containing captures, can be enclosed in a non-capturing group that uses mode modifiers. For example: `"^(?i:(?<prefix>...)bar)$"` will case-insensitively match any string that contains 6 characters, the last three of which are `"bar"`; the first three characters are stored in the named capture group `prefix` and are also available as capture group 1 within the pattern (the group containing the `?i:` is a non-capturing group).

In addition to the case-insensitive mode modifier, the *x mode modifier* may be used to enable *extended* mode. In extended mode, spaces within a pattern are ignored (they usually match literal space characters) as is everything from a `#` character to the end of the line. This mode can be useful for making complex patterns more readable by spacing them out across several lines and providing comments for individual portions of the pattern.

## 16.7 Debugging Queries

### 16.7.1 The assert Operator

The syntax for the `assert` operator is:

```
assert ( condition )
```

The `assert` operator takes a single parenthesized *condition* which is an arbitrary non-void expression. If the condition evaluates to `false` or `None`, error 340 is issued and the `assert` operator yields `false` after which processing continues normally, otherwise `assert` yields `true`.

### 16.7.2 Dumping the Query Tree

The `+dump_queries` option will cause PC-lint Plus to emit canonicalized query information along with corresponding Query AST immediately after processing subsequent `-astquery`, `-equery`, or `+equery` options. For example, the output generated for the option:

```
-astquery(VarDecl() : {
    !isLocalVarDeclOrParm() && getNameAsString().length() < 3
    message(8001 "variable name '" currentnode "' is too short")
})
```

will look something like:

```
Parsed AST Query:
  VarDecl() : {
    !isLocalVarDeclOrParm() && getNameAsString().length() < 3
    message(8001 "variable name '" currentnode "' is too short")
  }
Canonicalized Query:
  VarDecl(currentnode) : { not isLocalVarDeclOrParm(currentnode) and
    length(getNameAsString(currentnode)) < 3 message(8001 currentnode
    "variable name '" currentnode "' is too short") }
Query AST:
QueryContainer {Boolean} <Invalid location>
~-WithNode {Boolean} <<astquery option>:1:1-4:1>
|~-BuiltinFunctionCall VarDecl(ASTNode) {VarDecl} <<astquery option>:1:1-9>
| |~-Currentnode [Arg 1] {ASTNode} <Invalid location> (implicit)
|~-CompoundExpr {Boolean} <<astquery option>:1:13-4:1>
| |~-AndOperator {Boolean} <<astquery option>:2:5-61>
| | |~-NotOperator {Boolean} <<astquery option>:2:5-27>
| | | |~-BuiltinFunctionCall isLocalVarDeclOrParm(VarDecl) {Boolean} <<astquery option>:2:6-27>
| | | |~-Currentnode [Arg 1] {VarDecl} <Invalid location> (implicit)
| | | |~-LessThanOperator {Integral} <<astquery option>:2:50-61>
| | | | |~-BuiltinFunctionCall length(String) {Integral} <<astquery option>:2:50-55>
| | | | | |~-BuiltinFunctionCall getNameAsString(NamedDecl) [Arg 1] {String} <<astquery option>:2:32-48>
| | | | | |~-Currentnode [Arg 1] {VarDecl} <Invalid location> (implicit)
| | | | |~-Integer '3' {Integral} <<astquery option>:2:61>
|~-Message (8001) {Boolean} <<astquery option>:3:5-64>
| |~-Currentnode [Location] {VarDecl} <Invalid location> (implicit)
| |~-String "variable name '" [Message Parameter 1] {String} <<astquery option>:3:18-34>
| |~-Currentnode [Message Parameter 2] {VarDecl} <<astquery option>:3:36-46>
| |~-String "' is too short" [Message Parameter 3] {String} <<astquery option>:3:48-63>
```

The Query AST shows how the parsed query was interpreted. The static type of each query node is shown in braces after query node kind and possible annotation in square brackets. The location corresponding to the range of parsed text is shown in angle brackets. Some nodes have additional information shown in parentheses at the end of the node output.

The following sub-options may be used with `+dump_queries`: `unicode`, `nounicode`, `colors`, `nocolors`, `compact`, and `nocompact`. The first two sub-options specify whether Unicode line-drawing characters are used in the dumped AST. The next two options dictate whether portions of the AST are rendered in different colors. The last two sub-options determine whether the AST is horizontally compacted and is only supported when using Unicode line-drawing characters. The default mode on Linux and macOS is equivalent to `+dump_queries(unicode,colors,nocompact)`, the default mode on Windows is equivalent to `+dump_queries(nounicode,nocolors,nocompact)`.

The `-dump_queries` option suppresses AST query dumping for subsequently parsed queries. A later `+dump_queries` option will restore AST query dumping using the previously provided sub-options (if any). Multiple `+dump_queries` and `-dump_queries` may be used to cause a subset of query options to be affected.

## 16.8 Builtin Function List

### 16.8.1 Non-member functions

**getBoolType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin Boolean type.

**getChar16Type** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `char16_t` type in C++ mode and `unsigned short` in C mode.

**getChar32Type** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `char32_t` type in C++ mode and `unsigned int` in C mode.

**getChar8Type** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `char8_t` type.

**getCharType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `char` type.

**getCharWidthInBits** ()  $\Rightarrow$  *Integral*  
Returns the bit width of the `char` type.

**getDoubleType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `double` type.

**getFloat128Type** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `__float128` type.

**getFloatType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `float` type.

**getInt128Type** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `__int128` type.

**getIntMaxType** ()  $\Rightarrow$  *ASTType*  
Returns the `intmax_t` type.

**getIntPtrType** ()  $\Rightarrow$  *ASTType*  
Returns a type compatible with the `intptr_t` type.

**getIntType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `int` type.

**getLongDoubleType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `long double` type.

**getLongLongType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `long long` type.

**getLongType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `long` type.

**getNullPtrType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `nullptr_t` type.

**getPointerDiffType** ()  $\Rightarrow$  *ASTType*  
Returns the `ptrdiff_t` type.

**getShortType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `short` type.

**getSignedCharType** ()  $\Rightarrow$  *ASTType*  
Returns the builtin `signed char` type.

**getSignedIntTypeForBitwidth** (*Integral*)  $\Rightarrow$  *ASTType*  
Returns the builtin signed integer type with the specified bit size, if any.

**getSignedSizeType** ()  $\Rightarrow$  *ASTType*  
Returns the signed counterpart for the `size_t` type.

**getSizeType** ()  $\Rightarrow$  *ASTType*  
Returns the `size_t` type.

**getUIntMaxType** ()  $\Rightarrow$  *ASTType*  
Returns the `uintmax_t` type.

**getUIntPtrType** ()  $\Rightarrow$  *ASTType*  
Returns a type compatible with the `uintptr_t` type.

**getUnsignedCharType** ()  $\Rightarrow$  *ASTType*

- Returns the builtin `unsigned char` type.
- getUnsignedInt128Type** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `unsigned __int128` type.
- getUnsignedIntType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `unsigned int` type.
- getUnsignedIntTypeForBitwidth** (*Integral*)  $\Rightarrow$  *ASTType*
- Returns the builtin unsigned integer type with the specified bit size, if any.
- getUnsignedLongLongType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `unsigned long long` type.
- getUnsignedLongType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `unsigned long` type.
- getUnsignedPointerDiffType** ()  $\Rightarrow$  *ASTType*
- Returns the unsigned counterpart for the `ptrdiff_t` type.
- getUnsignedShortType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `unsigned short` type.
- getVoidPtrType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin pointer-to-void type.
- getVoidType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin `void` type.
- getWideCharType** ()  $\Rightarrow$  *ASTType*
- Returns the builtin wide character type.
- msgLocation** ()  $\Rightarrow$  *Location*
- Returns the primary location of the current message. This function is only available in `-equery/+equery` options.
- msgNumStringParams** ()  $\Rightarrow$  *Integral*
- Returns the number of *string* parameters in the current message. This function is only available in `-equery/+equery` options.
- msgNumSymbolParams** ()  $\Rightarrow$  *Integral*
- Returns the number of *symbol* parameters in the current message. This function is only available in `-equery/+equery` options.
- msgNumTypeParams** ()  $\Rightarrow$  *Integral*
- Returns the number of *type* parameters in the current message. This function is only available in `-equery/+equery` options.
- msgNumber** ()  $\Rightarrow$  *Integral*
- Returns the message number of the current message. This function is only available in `-equery/+equery` options.
- msgStringParam** (*Integral*)  $\Rightarrow$  *String*
- Returns the specified *string* parameter of the current message. This function is only available in `-equery/+equery` options.
- msgSymbolParam** (*Integral*)  $\Rightarrow$  *NamedDecl*
- Returns the specified *symbol* parameter of the current message. This function is only available in `-equery/+equery` options.
- msgTypeParam** (*Integral*)  $\Rightarrow$  *ASTType*
- Returns the specified *type* parameter of the current message. This function is only available in `-equery/+equery` options.
- threadID** ()  $\Rightarrow$  *Integral*
- Returns the numeric value of the running thread where 0 is the main thread (outside of a module) and a value of `n` represents the thread processing the `n`th source module.
- toBitsFromCharUnits** (*Integral*)  $\Rightarrow$  *Integral*
- Returns the number of bits needed to represent the provided number of `char` units.
- toCharUnitsFromBits** (*Integral*)  $\Rightarrow$  *Integral*
- Returns the number of `char` units needed to store the provided number of bits.

### 16.8.2 ASTNode functions

**AbstractConditionalOperator** (*ASTNode*)  $\Rightarrow$  *AbstractConditionalOperator*

If the provided AST node is convertible to an `AbstractConditionalOperator`, returns the node as an `AbstractConditionalOperator`, otherwise returns `None`.

**AccessSpecDecl** (*ASTNode*)  $\Rightarrow$  *AccessSpecDecl*

If the provided AST node is convertible to an *AccessSpecDecl*, returns the node as an *AccessSpecDecl*, otherwise returns *None*.

**AddrLabelExpr** (*ASTNode*)  $\Rightarrow$  *AddrLabelExpr*

If the provided AST node is convertible to an *AddrLabelExpr*, returns the node as an *AddrLabelExpr*, otherwise returns *None*.

**ArraySubscriptExpr** (*ASTNode*)  $\Rightarrow$  *ArraySubscriptExpr*

If the provided AST node is convertible to an *ArraySubscriptExpr*, returns the node as an *ArraySubscriptExpr*, otherwise returns *None*.

**AtomicExpr** (*ASTNode*)  $\Rightarrow$  *AtomicExpr*

If the provided AST node is convertible to an *AtomicExpr*, returns the node as an *AtomicExpr*, otherwise returns *None*.

**AttributedStmt** (*ASTNode*)  $\Rightarrow$  *AttributedStmt*

If the provided AST node is convertible to an *AttributedStmt*, returns the node as an *AttributedStmt*, otherwise returns *None*.

**BinaryConditionalOperator** (*ASTNode*)  $\Rightarrow$  *BinaryConditionalOperator*

If the provided AST node is convertible to a *BinaryConditionalOperator*, returns the node as a *BinaryConditionalOperator*, otherwise returns *None*.

**BinaryOperator** (*ASTNode*)  $\Rightarrow$  *BinaryOperator*

If the provided AST node is convertible to a *BinaryOperator*, returns the node as a *BinaryOperator*, otherwise returns *None*.

**BindingDecl** (*ASTNode*)  $\Rightarrow$  *BindingDecl*

If the provided AST node is convertible to a *BindingDecl*, returns the node as a *BindingDecl*, otherwise returns *None*.

**BreakStmt** (*ASTNode*)  $\Rightarrow$  *BreakStmt*

If the provided AST node is convertible to a *BreakStmt*, returns the node as a *BreakStmt*, otherwise returns *None*.

**CStyleCastExpr** (*ASTNode*)  $\Rightarrow$  *CStyleCastExpr*

If the provided AST node is convertible to a *CStyleCastExpr*, returns the node as a *CStyleCastExpr*, otherwise returns *None*.

**CXXBindTemporaryExpr** (*ASTNode*)  $\Rightarrow$  *CXXBindTemporaryExpr*

If the provided AST node is convertible to a *CXXBindTemporaryExpr*, returns the node as a *CXXBindTemporaryExpr*, otherwise returns *None*.

**CXXBoolLiteralExpr** (*ASTNode*)  $\Rightarrow$  *CXXBoolLiteralExpr*

If the provided AST node is convertible to a *CXXBoolLiteralExpr*, returns the node as a *CXXBoolLiteralExpr*, otherwise returns *None*.

**CXXCatchStmt** (*ASTNode*)  $\Rightarrow$  *CXXCatchStmt*

If the provided AST node is convertible to a *CXXCatchStmt*, returns the node as a *CXXCatchStmt*, otherwise returns *None*.

**CXXConstCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXConstCastExpr*

If the provided AST node is convertible to a *CXXConstCastExpr*, returns the node as a *CXXConstCastExpr*, otherwise returns *None*.

**CXXConstructExpr** (*ASTNode*)  $\Rightarrow$  *CXXConstructExpr*

If the provided AST node is convertible to a *CXXConstructExpr*, returns the node as a *CXXConstructExpr*, otherwise returns *None*.

**CXXConstructorDecl** (*ASTNode*)  $\Rightarrow$  *CXXConstructorDecl*

If the provided AST node is convertible to a *CXXConstructorDecl*, returns the node as a *CXXConstructorDecl*, otherwise returns *None*.

**CXXConversionDecl** (*ASTNode*)  $\Rightarrow$  *CXXConversionDecl*

If the provided AST node is convertible to a *CXXConversionDecl*, returns the node as a *CXXConversionDecl*, otherwise returns *None*.

**CXXDefaultArgExpr** (*ASTNode*)  $\Rightarrow$  *CXXDefaultArgExpr*

If the provided AST node is convertible to a *CXXDefaultArgExpr*, returns the node as a *CXXDefaultArgExpr*, otherwise returns *None*.

**CXXDefaultInitExpr** (*ASTNode*)  $\Rightarrow$  *CXXDefaultInitExpr*

If the provided AST node is convertible to a *CXXDefaultInitExpr*, returns the node as a *CXXDefaultInitExpr*, otherwise returns *None*.

**CXXDeleteExpr** (*ASTNode*)  $\Rightarrow$  *CXXDeleteExpr*

- If the provided AST node is convertible to a `CXXDeleteExpr`, returns the node as a `CXXDeleteExpr`, otherwise returns `None`.
- CXXDependentScopeMemberExpr** (*ASTNode*)  $\Rightarrow$  *CXXDependentScopeMemberExpr*  
 If the provided AST node is convertible to a `CXXDependentScopeMemberExpr`, returns the node as a `CXXDependentScopeMemberExpr`, otherwise returns `None`.
- CXXDestructorDecl** (*ASTNode*)  $\Rightarrow$  *CXXDestructorDecl*  
 If the provided AST node is convertible to a `CXXDestructorDecl`, returns the node as a `CXXDestructorDecl`, otherwise returns `None`.
- CXXDynamicCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXDynamicCastExpr*  
 If the provided AST node is convertible to a `CXXDynamicCastExpr`, returns the node as a `CXXDynamicCastExpr`, otherwise returns `None`.
- CXXFoldExpr** (*ASTNode*)  $\Rightarrow$  *CXXFoldExpr*  
 If the provided AST node is convertible to a `CXXFoldExpr`, returns the node as a `CXXFoldExpr`, otherwise returns `None`.
- CXXForRangeStmt** (*ASTNode*)  $\Rightarrow$  *CXXForRangeStmt*  
 If the provided AST node is convertible to a `CXXForRangeStmt`, returns the node as a `CXXForRangeStmt`, otherwise returns `None`.
- CXXFunctionalCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXFunctionalCastExpr*  
 If the provided AST node is convertible to a `CXXFunctionalCastExpr`, returns the node as a `CXXFunctionalCastExpr`, otherwise returns `None`.
- CXXInheritedCtorInitExpr** (*ASTNode*)  $\Rightarrow$  *CXXInheritedCtorInitExpr*  
 If the provided AST node is convertible to a `CXXInheritedCtorInitExpr`, returns the node as a `CXXInheritedCtorInitExpr`, otherwise returns `None`.
- CXXMemberCallExpr** (*ASTNode*)  $\Rightarrow$  *CXXMemberCallExpr*  
 If the provided AST node is convertible to a `CXXMemberCallExpr`, returns the node as a `CXXMemberCallExpr`, otherwise returns `None`.
- CXXMethodDecl** (*ASTNode*)  $\Rightarrow$  *CXXMethodDecl*  
 If the provided AST node is convertible to a `CXXMethodDecl`, returns the node as a `CXXMethodDecl`, otherwise returns `None`.
- CXXNamedCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXNamedCastExpr*  
 If the provided AST node is convertible to a `CXXNamedCastExpr`, returns the node as a `CXXNamedCastExpr`, otherwise returns `None`.
- CXXNewExpr** (*ASTNode*)  $\Rightarrow$  *CXXNewExpr*  
 If the provided AST node is convertible to a `CXXNewExpr`, returns the node as a `CXXNewExpr`, otherwise returns `None`.
- CXXNoexceptExpr** (*ASTNode*)  $\Rightarrow$  *CXXNoexceptExpr*  
 If the provided AST node is convertible to a `CXXNoexceptExpr`, returns the node as a `CXXNoexceptExpr`, otherwise returns `None`.
- CXXNullPtrLiteralExpr** (*ASTNode*)  $\Rightarrow$  *CXXNullPtrLiteralExpr*  
 If the provided AST node is convertible to a `CXXNullPtrLiteralExpr`, returns the node as a `CXXNullPtrLiteralExpr`, otherwise returns `None`.
- CXXOperatorCallExpr** (*ASTNode*)  $\Rightarrow$  *CXXOperatorCallExpr*  
 If the provided AST node is convertible to a `CXXOperatorCallExpr`, returns the node as a `CXXOperatorCallExpr`, otherwise returns `None`.
- CXXPseudoDestructorExpr** (*ASTNode*)  $\Rightarrow$  *CXXPseudoDestructorExpr*  
 If the provided AST node is convertible to a `CXXPseudoDestructorExpr`, returns the node as a `CXXPseudoDestructorExpr`, otherwise returns `None`.
- CXXRecordDecl** (*ASTNode*)  $\Rightarrow$  *CXXRecordDecl*  
 If the provided AST node is convertible to a `CXXRecordDecl`, returns the node as a `CXXRecordDecl`, otherwise returns `None`.
- CXXReinterpretCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXReinterpretCastExpr*  
 If the provided AST node is convertible to a `CXXReinterpretCastExpr`, returns the node as a `CXXReinterpretCastExpr`, otherwise returns `None`.
- CXXScalarValueInitExpr** (*ASTNode*)  $\Rightarrow$  *CXXScalarValueInitExpr*  
 If the provided AST node is convertible to a `CXXScalarValueInitExpr`, returns the node as a `CXXScalarValueInitExpr`, otherwise returns `None`.



**CXXStaticCastExpr** (*ASTNode*)  $\Rightarrow$  *CXXStaticCastExpr*

If the provided AST node is convertible to a *CXXStaticCastExpr*, returns the node as a *CXXStaticCastExpr*, otherwise returns *None*.

**CXXStdInitializerListExpr** (*ASTNode*)  $\Rightarrow$  *CXXStdInitializerListExpr*

If the provided AST node is convertible to a *CXXStdInitializerListExpr*, returns the node as a *CXXStdInitializerListExpr*, otherwise returns *None*.

**CXXThisExpr** (*ASTNode*)  $\Rightarrow$  *CXXThisExpr*

If the provided AST node is convertible to a *CXXThisExpr*, returns the node as a *CXXThisExpr*, otherwise returns *None*.

**CXXThrowExpr** (*ASTNode*)  $\Rightarrow$  *CXXThrowExpr*

If the provided AST node is convertible to a *CXXThrowExpr*, returns the node as a *CXXThrowExpr*, otherwise returns *None*.

**CXXTryStmt** (*ASTNode*)  $\Rightarrow$  *CXXTryStmt*

If the provided AST node is convertible to a *CXXTryStmt*, returns the node as a *CXXTryStmt*, otherwise returns *None*.

**CXXTypeidExpr** (*ASTNode*)  $\Rightarrow$  *CXXTypeidExpr*

If the provided AST node is convertible to a *CXXTypeidExpr*, returns the node as a *CXXTypeidExpr*, otherwise returns *None*.

**CallExpr** (*ASTNode*)  $\Rightarrow$  *CallExpr*

If the provided AST node is convertible to a *CallExpr*, returns the node as a *CallExpr*, otherwise returns *None*.

**CaseStmt** (*ASTNode*)  $\Rightarrow$  *CaseStmt*

If the provided AST node is convertible to a *CaseStmt*, returns the node as a *CaseStmt*, otherwise returns *None*.

**CastExpr** (*ASTNode*)  $\Rightarrow$  *CastExpr*

If the provided AST node is convertible to a *CastExpr*, returns the node as a *CastExpr*, otherwise returns *None*.

**CharacterLiteral** (*ASTNode*)  $\Rightarrow$  *CharacterLiteral*

If the provided AST node is convertible to a *CharacterLiteral*, returns the node as a *CharacterLiteral*, otherwise returns *None*.

**ClassTemplateDecl** (*ASTNode*)  $\Rightarrow$  *ClassTemplateDecl*

If the provided AST node is convertible to a *ClassTemplateDecl*, returns the node as a *ClassTemplateDecl*, otherwise returns *None*.

**ClassTemplatePartialSpecializationDecl** (*ASTNode*)  $\Rightarrow$  *ClassTemplatePartialSpecializationDecl*

If the provided AST node is convertible to a *ClassTemplatePartialSpecializationDecl*, returns the node as a *ClassTemplatePartialSpecializationDecl*, otherwise returns *None*.

**ClassTemplateSpecializationDecl** (*ASTNode*)  $\Rightarrow$  *ClassTemplateSpecializationDecl*

If the provided AST node is convertible to a *ClassTemplateSpecializationDecl*, returns the node as a *ClassTemplateSpecializationDecl*, otherwise returns *None*.

**CompoundAssignOperator** (*ASTNode*)  $\Rightarrow$  *CompoundAssignOperator*

If the provided AST node is convertible to a *CompoundAssignOperator*, returns the node as a *CompoundAssignOperator*, otherwise returns *None*.

**CompoundLiteralExpr** (*ASTNode*)  $\Rightarrow$  *CompoundLiteralExpr*

If the provided AST node is convertible to a *CompoundLiteralExpr*, returns the node as a *CompoundLiteralExpr*, otherwise returns *None*.

**CompoundStmt** (*ASTNode*)  $\Rightarrow$  *CompoundStmt*

If the provided AST node is convertible to a *CompoundStmt*, returns the node as a *CompoundStmt*, otherwise returns *None*.

**ConditionalOperator** (*ASTNode*)  $\Rightarrow$  *ConditionalOperator*

If the provided AST node is convertible to a *ConditionalOperator*, returns the node as a *ConditionalOperator*, otherwise returns *None*.

**ContinueStmt** (*ASTNode*)  $\Rightarrow$  *ContinueStmt*

If the provided AST node is convertible to a *ContinueStmt*, returns the node as a *ContinueStmt*, otherwise returns *None*.

**CoreturnStmt** (*ASTNode*)  $\Rightarrow$  *CoreturnStmt*

If the provided AST node is convertible to a *CoreturnStmt*, returns the node as a *CoreturnStmt*, otherwise returns *None*.

**CoroutineBodyStmt** (*ASTNode*)  $\Rightarrow$  *CoroutineBodyStmt*

If the provided AST node is convertible to a *CoroutineBodyStmt*, returns the node as a *CoroutineBodyStmt*, otherwise returns *None*.

**Decl** (*ASTNode*)  $\Rightarrow$  *Decl*

If the provided AST node is convertible to a *Decl*, returns the node as a *Decl*, otherwise returns *None*.

**DeclRefExpr** (*ASTNode*)  $\Rightarrow$  *DeclRefExpr*

If the provided AST node is convertible to a `DeclRefExpr`, returns the node as a `DeclRefExpr`, otherwise returns `None`.

**DeclStmt** (*ASTNode*)  $\Rightarrow$  *DeclStmt*

If the provided AST node is convertible to a `DeclStmt`, returns the node as a `DeclStmt`, otherwise returns `None`.

**DeclaratorDecl** (*ASTNode*)  $\Rightarrow$  *DeclaratorDecl*

If the provided AST node is convertible to a `DeclaratorDecl`, returns the node as a `DeclaratorDecl`, otherwise returns `None`.

**DecompositionDecl** (*ASTNode*)  $\Rightarrow$  *DecompositionDecl*

If the provided AST node is convertible to a `DecompositionDecl`, returns the node as a `DecompositionDecl`, otherwise returns `None`.

**DefaultStmt** (*ASTNode*)  $\Rightarrow$  *DefaultStmt*

If the provided AST node is convertible to a `DefaultStmt`, returns the node as a `DefaultStmt`, otherwise returns `None`.

**DesignatedInitExpr** (*ASTNode*)  $\Rightarrow$  *DesignatedInitExpr*

If the provided AST node is convertible to a `DesignatedInitExpr`, returns the node as a `DesignatedInitExpr`, otherwise returns `None`.

**DoStmt** (*ASTNode*)  $\Rightarrow$  *DoStmt*

If the provided AST node is convertible to a `DoStmt`, returns the node as a `DoStmt`, otherwise returns `None`.

**EmptyDecl** (*ASTNode*)  $\Rightarrow$  *EmptyDecl*

If the provided AST node is convertible to an `EmptyDecl`, returns the node as an `EmptyDecl`, otherwise returns `None`.

**EnumConstantDecl** (*ASTNode*)  $\Rightarrow$  *EnumConstantDecl*

If the provided AST node is convertible to an `EnumConstantDecl`, returns the node as an `EnumConstantDecl`, otherwise returns `None`.

**EnumDecl** (*ASTNode*)  $\Rightarrow$  *EnumDecl*

If the provided AST node is convertible to an `EnumDecl`, returns the node as an `EnumDecl`, otherwise returns `None`.

**ExplicitCastExpr** (*ASTNode*)  $\Rightarrow$  *ExplicitCastExpr*

If the provided AST node is convertible to an `ExplicitCastExpr`, returns the node as an `ExplicitCastExpr`, otherwise returns `None`.

**Expr** (*ASTNode*)  $\Rightarrow$  *Expr*

If the provided AST node is convertible to an `Expr`, returns the node as an `Expr`, otherwise returns `None`.

**ExternCContextDecl** (*ASTNode*)  $\Rightarrow$  *ExternCContextDecl*

If the provided AST node is convertible to an `ExternCContextDecl`, returns the node as an `ExternCContextDecl`, otherwise returns `None`.

**FieldDecl** (*ASTNode*)  $\Rightarrow$  *FieldDecl*

If the provided AST node is convertible to a `FieldDecl`, returns the node as a `FieldDecl`, otherwise returns `None`.

**FloatingLiteral** (*ASTNode*)  $\Rightarrow$  *FloatingLiteral*

If the provided AST node is convertible to a `FloatingLiteral`, returns the node as a `FloatingLiteral`, otherwise returns `None`.

**ForStmt** (*ASTNode*)  $\Rightarrow$  *ForStmt*

If the provided AST node is convertible to a `ForStmt`, returns the node as a `ForStmt`, otherwise returns `None`.

**FriendDecl** (*ASTNode*)  $\Rightarrow$  *FriendDecl*

If the provided AST node is convertible to a `FriendDecl`, returns the node as a `FriendDecl`, otherwise returns `None`.

**FriendTemplateDecl** (*ASTNode*)  $\Rightarrow$  *FriendTemplateDecl*

If the provided AST node is convertible to a `FriendTemplateDecl`, returns the node as a `FriendTemplateDecl`, otherwise returns `None`.

**FunctionDecl** (*ASTNode*)  $\Rightarrow$  *FunctionDecl*

If the provided AST node is convertible to a `FunctionDecl`, returns the node as a `FunctionDecl`, otherwise returns `None`.

**FunctionTemplateDecl** (*ASTNode*)  $\Rightarrow$  *FunctionTemplateDecl*

If the provided AST node is convertible to a `FunctionTemplateDecl`, returns the node as a `FunctionTemplateDecl`, otherwise returns `None`.

**GNUNullExpr** (*ASTNode*)  $\Rightarrow$  *GNUNullExpr*

If the provided AST node is convertible to a `GNUNullExpr`, returns the node as a `GNUNullExpr`, otherwise returns `None`.

**GenericSelectionExpr** (*ASTNode*)  $\Rightarrow$  *GenericSelectionExpr*

If the provided AST node is convertible to a `GenericSelectionExpr`, returns the node as a `GenericSelectionExpr`, otherwise returns `None`.

**GotoStmt** (*ASTNode*)  $\Rightarrow$  *GotoStmt*

- If the provided AST node is convertible to a `GotoStmt`, returns the node as a `GotoStmt`, otherwise returns `None`.
- IfStmt** (*ASTNode*)  $\Rightarrow$  *IfStmt*  
 If the provided AST node is convertible to an `IfStmt`, returns the node as an `IfStmt`, otherwise returns `None`.
- ImaginaryLiteral** (*ASTNode*)  $\Rightarrow$  *ImaginaryLiteral*  
 If the provided AST node is convertible to an `ImaginaryLiteral`, returns the node as an `ImaginaryLiteral`, otherwise returns `None`.
- ImplicitCastExpr** (*ASTNode*)  $\Rightarrow$  *ImplicitCastExpr*  
 If the provided AST node is convertible to an `ImplicitCastExpr`, returns the node as an `ImplicitCastExpr`, otherwise returns `None`.
- ImplicitParamDecl** (*ASTNode*)  $\Rightarrow$  *ImplicitParamDecl*  
 If the provided AST node is convertible to an `ImplicitParamDecl`, returns the node as an `ImplicitParamDecl`, otherwise returns `None`.
- IndirectFieldDecl** (*ASTNode*)  $\Rightarrow$  *IndirectFieldDecl*  
 If the provided AST node is convertible to an `IndirectFieldDecl`, returns the node as an `IndirectFieldDecl`, otherwise returns `None`.
- InitListExpr** (*ASTNode*)  $\Rightarrow$  *InitListExpr*  
 If the provided AST node is convertible to an `InitListExpr`, returns the node as an `InitListExpr`, otherwise returns `None`.
- IntegerLiteral** (*ASTNode*)  $\Rightarrow$  *IntegerLiteral*  
 If the provided AST node is convertible to an `IntegerLiteral`, returns the node as an `IntegerLiteral`, otherwise returns `None`.
- LabelDecl** (*ASTNode*)  $\Rightarrow$  *LabelDecl*  
 If the provided AST node is convertible to a `LabelDecl`, returns the node as a `LabelDecl`, otherwise returns `None`.
- LabelStmt** (*ASTNode*)  $\Rightarrow$  *LabelStmt*  
 If the provided AST node is convertible to a `LabelStmt`, returns the node as a `LabelStmt`, otherwise returns `None`.
- LambdaExpr** (*ASTNode*)  $\Rightarrow$  *LambdaExpr*  
 If the provided AST node is convertible to a `LambdaExpr`, returns the node as a `LambdaExpr`, otherwise returns `None`.
- LinkageSpecDecl** (*ASTNode*)  $\Rightarrow$  *LinkageSpecDecl*  
 If the provided AST node is convertible to a `LinkageSpecDecl`, returns the node as a `LinkageSpecDecl`, otherwise returns `None`.
- MemberExpr** (*ASTNode*)  $\Rightarrow$  *MemberExpr*  
 If the provided AST node is convertible to a `MemberExpr`, returns the node as a `MemberExpr`, otherwise returns `None`.
- NamedDecl** (*ASTNode*)  $\Rightarrow$  *NamedDecl*  
 If the provided AST node is convertible to a `NamedDecl`, returns the node as a `NamedDecl`, otherwise returns `None`.
- NamespaceAliasDecl** (*ASTNode*)  $\Rightarrow$  *NamespaceAliasDecl*  
 If the provided AST node is convertible to a `NamespaceAliasDecl`, returns the node as a `NamespaceAliasDecl`, otherwise returns `None`.
- NamespaceDecl** (*ASTNode*)  $\Rightarrow$  *NamespaceDecl*  
 If the provided AST node is convertible to a `NamespaceDecl`, returns the node as a `NamespaceDecl`, otherwise returns `None`.
- NullStmt** (*ASTNode*)  $\Rightarrow$  *NullStmt*  
 If the provided AST node is convertible to a `NullStmt`, returns the node as a `NullStmt`, otherwise returns `None`.
- OffsetOfExpr** (*ASTNode*)  $\Rightarrow$  *OffsetOfExpr*  
 If the provided AST node is convertible to an `OffsetOfExpr`, returns the node as an `OffsetOfExpr`, otherwise returns `None`.
- ParenExpr** (*ASTNode*)  $\Rightarrow$  *ParenExpr*  
 If the provided AST node is convertible to a `ParenExpr`, returns the node as a `ParenExpr`, otherwise returns `None`.
- ParmVarDecl** (*ASTNode*)  $\Rightarrow$  *ParmVarDecl*  
 If the provided AST node is convertible to a `ParmVarDecl`, returns the node as a `ParmVarDecl`, otherwise returns `None`.
- RecordDecl** (*ASTNode*)  $\Rightarrow$  *RecordDecl*  
 If the provided AST node is convertible to a `RecordDecl`, returns the node as a `RecordDecl`, otherwise returns `None`.
- RedeclarableTemplateDecl** (*ASTNode*)  $\Rightarrow$  *RedeclarableTemplateDecl*  
 If the provided AST node is convertible to a `RedeclarableTemplateDecl`, returns the node as a `RedeclarableTemplateDecl`, otherwise returns `None`.

**ReturnStmt** (*ASTNode*)  $\Rightarrow$  *ReturnStmt*

If the provided AST node is convertible to a **ReturnStmt**, returns the node as a **ReturnStmt**, otherwise returns **None**.

**StaticAssertDecl** (*ASTNode*)  $\Rightarrow$  *StaticAssertDecl*

If the provided AST node is convertible to a **StaticAssertDecl**, returns the node as a **StaticAssertDecl**, otherwise returns **None**.

**Stmt** (*ASTNode*)  $\Rightarrow$  *Stmt*

If the provided AST node is convertible to a **Stmt**, returns the node as a **Stmt**, otherwise returns **None**.

**StmtExpr** (*ASTNode*)  $\Rightarrow$  *StmtExpr*

If the provided AST node is convertible to a **StmtExpr**, returns the node as a **StmtExpr**, otherwise returns **None**.

**StringLiteral** (*ASTNode*)  $\Rightarrow$  *StringLiteral*

If the provided AST node is convertible to a **StringLiteral**, returns the node as a **StringLiteral**, otherwise returns **None**.

**SwitchCase** (*ASTNode*)  $\Rightarrow$  *SwitchCase*

If the provided AST node is convertible to a **SwitchCase**, returns the node as a **SwitchCase**, otherwise returns **None**.

**SwitchStmt** (*ASTNode*)  $\Rightarrow$  *SwitchStmt*

If the provided AST node is convertible to a **SwitchStmt**, returns the node as a **SwitchStmt**, otherwise returns **None**.

**TagDecl** (*ASTNode*)  $\Rightarrow$  *TagDecl*

If the provided AST node is convertible to a **TagDecl**, returns the node as a **TagDecl**, otherwise returns **None**.

**TemplateDecl** (*ASTNode*)  $\Rightarrow$  *TemplateDecl*

If the provided AST node is convertible to a **TemplateDecl**, returns the node as a **TemplateDecl**, otherwise returns **None**.

**TemplateTypeParmDecl** (*ASTNode*)  $\Rightarrow$  *TemplateTypeParmDecl*

If the provided AST node is convertible to a **TemplateTypeParmDecl**, returns the node as a **TemplateTypeParmDecl**, otherwise returns **None**.

**TranslationUnitDecl** (*ASTNode*)  $\Rightarrow$  *TranslationUnitDecl*

If the provided AST node is convertible to a **TranslationUnitDecl**, returns the node as a **TranslationUnitDecl**, otherwise returns **None**.

**TypeAliasDecl** (*ASTNode*)  $\Rightarrow$  *TypeAliasDecl*

If the provided AST node is convertible to a **TypeAliasDecl**, returns the node as a **TypeAliasDecl**, otherwise returns **None**.

**TypeAliasTemplateDecl** (*ASTNode*)  $\Rightarrow$  *TypeAliasTemplateDecl*

If the provided AST node is convertible to a **TypeAliasTemplateDecl**, returns the node as a **TypeAliasTemplateDecl**, otherwise returns **None**.

**TypeDecl** (*ASTNode*)  $\Rightarrow$  *TypeDecl*

If the provided AST node is convertible to a **TypeDecl**, returns the node as a **TypeDecl**, otherwise returns **None**.

**TypedefDecl** (*ASTNode*)  $\Rightarrow$  *TypedefDecl*

If the provided AST node is convertible to a **TypedefDecl**, returns the node as a **TypedefDecl**, otherwise returns **None**.

**TypedefNameDecl** (*ASTNode*)  $\Rightarrow$  *TypedefNameDecl*

If the provided AST node is convertible to a **TypedefNameDecl**, returns the node as a **TypedefNameDecl**, otherwise returns **None**.

**UnaryExprOrTypeTraitExpr** (*ASTNode*)  $\Rightarrow$  *UnaryExprOrTypeTraitExpr*

If the provided AST node is convertible to a **UnaryExprOrTypeTraitExpr**, returns the node as a **UnaryExprOrTypeTraitExpr**, otherwise returns **None**.

**UnaryOperator** (*ASTNode*)  $\Rightarrow$  *UnaryOperator*

If the provided AST node is convertible to a **UnaryOperator**, returns the node as a **UnaryOperator**, otherwise returns **None**.

**UserDefinedLiteral** (*ASTNode*)  $\Rightarrow$  *UserDefinedLiteral*

If the provided AST node is convertible to a **UserDefinedLiteral**, returns the node as a **UserDefinedLiteral**, otherwise returns **None**.

**UsingDecl** (*ASTNode*)  $\Rightarrow$  *UsingDecl*

If the provided AST node is convertible to a **UsingDecl**, returns the node as a **UsingDecl**, otherwise returns **None**.

**UsingDirectiveDecl** (*ASTNode*)  $\Rightarrow$  *UsingDirectiveDecl*

If the provided AST node is convertible to a **UsingDirectiveDecl**, returns the node as a **UsingDirectiveDecl**, otherwise returns **None**.

**UsingShadowDecl** (*ASTNode*)  $\Rightarrow$  *UsingShadowDecl*

If the provided AST node is convertible to a `UsingShadowDecl`, returns the node as a `UsingShadowDecl`, otherwise returns `None`.

**ValueDecl** (*ASTNode*)  $\Rightarrow$  *ValueDecl*  
 If the provided AST node is convertible to a `ValueDecl`, returns the node as a `ValueDecl`, otherwise returns `None`.

**ValueStmt** (*ASTNode*)  $\Rightarrow$  *ValueStmt*  
 If the provided AST node is convertible to a `ValueStmt`, returns the node as a `ValueStmt`, otherwise returns `None`.

**VarDecl** (*ASTNode*)  $\Rightarrow$  *VarDecl*  
 If the provided AST node is convertible to a `VarDecl`, returns the node as a `VarDecl`, otherwise returns `None`.

**VarTemplateDecl** (*ASTNode*)  $\Rightarrow$  *VarTemplateDecl*  
 If the provided AST node is convertible to a `VarTemplateDecl`, returns the node as a `VarTemplateDecl`, otherwise returns `None`.

**VarTemplatePartialSpecializationDecl** (*ASTNode*)  $\Rightarrow$  *VarTemplatePartialSpecializationDecl*  
 If the provided AST node is convertible to a `VarTemplatePartialSpecializationDecl`, returns the node as a `VarTemplatePartialSpecializationDecl`, otherwise returns `None`.

**VarTemplateSpecializationDecl** (*ASTNode*)  $\Rightarrow$  *VarTemplateSpecializationDecl*  
 If the provided AST node is convertible to a `VarTemplateSpecializationDecl`, returns the node as a `VarTemplateSpecializationDecl`, otherwise returns `None`.

**WhileStmt** (*ASTNode*)  $\Rightarrow$  *WhileStmt*  
 If the provided AST node is convertible to a `WhileStmt`, returns the node as a `WhileStmt`, otherwise returns `None`.

**getLocation** (*ASTNode*)  $\Rightarrow$  *Location*  
 Returns the location of the provided AST node.

### 16.8.3 ASTType functions

**canDecayToPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
 Returns true when `isFunctionType` or `isArrayType` would return true for the provided type.

**getArrayType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 If the provided type is an array type, returns the type of the array's elements, otherwise returns `None`.

**getArraySizeExpr** (*ASTType*)  $\Rightarrow$  *Expr*  
 If the provided type is a constant, dependent, or variable sized array, returns the expression that represents the array's size, otherwise returns `None`.

**getArraySizeModifier** (*ASTType*)  $\Rightarrow$  *String*  
 If the provided type is an array type, returns a string (one of "Normal", "Static", or "Star") that represents the array's size modifier, otherwise returns `None`.

**getAsCXXRecordDecl** (*ASTType*)  $\Rightarrow$  *CXXRecordDecl*  
 If the provided type is a `CXXRecordDecl`, returns the corresponding declaration, otherwise returns `None`.

**getAsRecordDecl** (*ASTType*)  $\Rightarrow$  *RecordDecl*  
 If the provided type is a `RecordDecl`, returns the corresponding declaration, otherwise returns `None`.

**getAsString** (*ASTType*)  $\Rightarrow$  *String*  
 Returns a string representation of the provided type.

**getAsTagDecl** (*ASTType*)  $\Rightarrow$  *TagDecl*  
 If the provided type is a `TagDecl`, returns the corresponding declaration, otherwise returns `None`.

**getAtomicUnqualifiedType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 Returns the provided type with `const`, `volatile`, `restrict`, and `_Atomic` qualifier removed.

**getCanonicalType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 Returns the canonical type of the provided type.

**getConstantArraySize** (*ASTType*)  $\Rightarrow$  *Integral*  
 If the provided type is a constant sized array, return the size value of the array, otherwise returns `None`.

**getFunctionCallResultType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 If the provided type is a function type, returns the function's call result type, otherwise returns `None`.

**getFunctionReturnType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 If the provided type is a function type, returns the function's return type, otherwise returns `None`.

**getNonReferenceType** (*ASTType*)  $\Rightarrow$  *ASTType*  
 If the provided type is a reference type, returns the type to which the reference refers, otherwise returns the provided type.

**getPointeeCXXRecordDecl** (*ASTType*)  $\Rightarrow$  *CXXRecordDecl*

If the provided type is a pointer or reference to a *CXXRecordDecl*, returns the corresponding *CXXRecordDecl*, otherwise returns *None*.

**getPointeeType** (*ASTType*)  $\Rightarrow$  *ASTType*

If the provided type is a pointer, pointer to member, or reference type, returns the corresponding pointee type, otherwise returns *None*.

**getTypeSizeInBits** (*ASTType*)  $\Rightarrow$  *Integral*

Returns size of the provided type, if known, in bits.

**getTypeSizeInChars** (*ASTType*)  $\Rightarrow$  *Integral*

Returns size of the provided type, if known, in *char* units.

**hasLocalQualifiers** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is locally qualified.

**hasQualifiers** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is *const*, *restrict*, or *volatile* qualified.

**isAggregateType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true the provided type is a C++ aggregate type or a C aggregate or union type.

**isAnyCharacterType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if *isCharType*, *isWideCharType*, *isChar8Type*, *isChar16Type*, or *isChar32Type* would return true for the provided type.

**isAnyComplexType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an integral or floating point complex type.

**isArithmeticType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true when either *isRealType* or *isAnyComplexType* would return true.

**isArrayIndexTypeConstQualified** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an array type with a *const* qualified index type such as the type returns by *getOriginalType* for the *ParmVarDecl* for the parameter in a function declared as `void foo(int a[const n]);`.

**isArrayIndexTypeRestrictQualified** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an array type with a *restrict* qualified index type such as the type returns by *getOriginalType* for the *ParmVarDecl* for the parameter in a function declared as `void foo(int a[restrict n]);`.

**isArrayIndexTypeVolatileQualified** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an array type with a *volatile* qualified index type such as the type returns by *getOriginalType* for the *ParmVarDecl* for the parameter in a function declared as `void foo(int a[volatile n]);`.

**isArrayType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an array type.

**isAtLeastAsQualifiedAs** (*ASTType* *ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the first provided type is at least as qualified as the second provided type.

**isAtomicType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a C11 atomic type declared with *\_Atomic*.

**isBooleanType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a Boolean type.

**isBuiltinType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type corresponds to a builtin type.

**isCForbiddenLValueType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is the unqualified void type or a function type.

**isCXX11PODType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a POD type according to the rules of the C++ 11 standard.

**isCXX98PODType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a POD type according to the rules of the C++ 98 standard.

**isChar16Type** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the canonical type of the provided type is *char16\_t*.

**isChar32Type** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the canonical type of the provided type is *char32\_t*.

**isChar8Type** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the canonical type of the provided type is *char8\_t*.

**isCharType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the canonical type of the provided type is one of `char`, `signed char`, or `unsigned char`.

**isClassType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a class type, declared with the `class` keyword.

**isComplexType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a complex floating point type.

**isCompoundType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a *compound* type, i.e. if any of `isArrayType`, `isFunctionType`, `isPointerType`, `isReferenceType`, `isRecordType`, `isUnionType`, `isEnumeralType`, or `isMemberPointerType` would return true for the provided type.

**isConstQualified** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is `const` qualified.

**isConstantArrayType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a constant sized array type (e.g. `int array[10];`).

**isConstantSizeType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is not an incomplete, dependent, or variable-sized type.

**isDependentSizedArrayType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a dependent sized array type (e.g. `int array[T];`).

**isDependentType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the definition of the provided type depends on a template parameter.

**isElaboratedTypeSpecifier** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a C++ elaborated-type-specifier.

**isEnumeralType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an enumeral type.

**isFloatingType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a real or complex floating point type.

**isFunctionNoProtoType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a function type without parameter type information.

**isFunctionPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a function pointer type.

**isFunctionProtoType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a function type with parameter type information.

**isFunctionType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a function type.

**isFundamentalType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a *fundamental* type, i.e. if `isVoidType`, `isNullPtrType`, or `isArithmeticType` would return true for the provided type and `isEnumeralType` type would return false.

**isIncompleteArrayType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an incomplete array type (e.g. `int array[];`).

**isIncompleteOrObjectType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an incomplete or object type, i.e. not a function type.

**isIncompleteType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is an incomplete type, i.e. not an object or function type.

**isInstantiationDependentType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type involves a template parameter, regardless of whether its definition depends on template parameter.

**isIntegerType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type is a (non-complex) integer type.

**isIntegralOrEnumerationType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type has integral or enumeration type.

**isIntegralOrUnscopedEnumerationType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type has integral or unscoped enumeration type.

**isIntegralType** (*ASTType*)  $\Rightarrow$  *Boolean*

Returns true if the provided type has integral type.

- isInterfaceType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is an interface type.
- isLValueReferenceType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a lvalue-reference type.
- isLiteralType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a C++ 11 literal type.
- isLocalConstQualified** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is locally `const` qualified.
- isLocalRestrictQualified** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is locally `restrict` qualified.
- isLocalVolatileQualified** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is locally `volatile` qualified.
- isMemberDataPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a pointer to data member type.
- isMemberFunctionPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a pointer to function member type.
- isMemberPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true when either `isMemberFunctionPointerType` or `isMemberDataPointerType` would return true.
- isMoreQualifiedThan** (*ASTType* *ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the first provided type is at more as qualified as the second provided type.
- isNothrowT** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is the `std::nothrow_t` type.
- isNullPtrType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is the `std::nullptr_t` type.
- isObjectType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is an object type, i.e. not an incomplete or function type.
- isOverloadableType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true when `isDependentType`, `isRecordType`, or `isEnumeralType` would return true for the provided type.
- isPODType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a POD type according to the current C++ language mode.
- isPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a pointer type.
- isPromotableIntegerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is promotable.
- isRValueReferenceType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a rvalue-reference type.
- isRealFloatingType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a real (i.e. not complex) floating point type (e.g. `float`, `double`, etc.).
- isRealType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a real (not complex) floating point or integral type or a non-scoped, complete, enumeration type.
- isRecordType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true when the `isStructureOrClassType` or `isUnionType` functions would return true.
- isReferenceType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a reference type.
- isRestrictQualified** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is `restrict` qualified.
- isScalarType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a scalar type.
- isScopedEnumeralType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a scoped enumeration type.
- isSignedIntegerOrEnumerationType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a signed integer type or has enumeration type with an underlying signed integer type.
- isSignedIntegerType** (*ASTType*)  $\Rightarrow$  *Boolean*



- Returns true if the provided type is a signed integer type or has unscoped enumeration type with an underlying signed integer type.
- isStandardLayoutType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a standard-layout type.
- isStdByteType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is the `std::byte` type.
- isStdInitList** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is an instantiation of `std::initializer_list`.
- isStructuralType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a C++ 20 structural type.
- isStructureOrClassType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true when the `isClassType`, `isStructureType`, or `isInterfaceType` functions would return true.
- isStructureType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a structure or class type, declared with the `struct` keyword.
- isTemplateTypeParmType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a template type parameter type.
- isTrivialType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a trivial type.
- isTriviallyCopyableType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is trivially copyable.
- isTypedefNameType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type was written as a typedef name.
- isUnionType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a `union` type.
- isUnscopedEnumerationType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a non-scoped enumeration type.
- isUnsignedIntegerOrEnumerationType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is an unsigned integer type or has enumeration type with an underlying unsigned integer type.
- isUnsignedIntegerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is an unsigned integer type or has unscoped enumeration type with an underlying unsigned integer type.
- isVariableArrayType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a variably sized array type (e.g. `int array[n];`).
- isVariablyModifiedType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if this is a C99 variably modified type (the type of a variable length array).
- isVoidPointerType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a void pointer type.
- isVoidType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is a void type.
- isVolatileQualified** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the provided type is volatile qualified.
- isWideCharType** (*ASTType*)  $\Rightarrow$  *Boolean*  
Returns true if the canonical type of the provided type is `wchar_t`.

#### 16.8.4 ArraySubscriptExpr functions

- getBase** (*ArraySubscriptExpr*)  $\Rightarrow$  *Expr*  
Returns the expression that forms the base of the provided array subscript expression, regardless of operand order.
- getIdx** (*ArraySubscriptExpr*)  $\Rightarrow$  *Expr*  
Returns the expression that forms the index of the provided array subscript expression, regardless of operand order.
- getLHS** (*ArraySubscriptExpr*)  $\Rightarrow$  *Expr*  
Returns the expression appearing on the LHS of the provided array subscript expression.
- getRHS** (*ArraySubscriptExpr*)  $\Rightarrow$  *Expr*

Returns the expression appearing on the RHS of the provided array subscript expression, i.e. the expression enclosed by square brackets.

### 16.8.5 AttributedStmt functions

**getSubStmt** (*AttributedStmt*)  $\Rightarrow$  *Stmt*

Returns the substatement of the provided attributed statement.

### 16.8.6 BinaryOperator functions

**getLHS** (*BinaryOperator*)  $\Rightarrow$  *Expr*

Returns the LHS operand of the provided binary operator.

**getOpcodeStr** (*BinaryOperator*)  $\Rightarrow$  *String*

Returns the string representation of the binary operator.

**getRHS** (*BinaryOperator*)  $\Rightarrow$  *Expr*

Returns the RHS operand of the provided binary operator.

**isAdditiveOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is an additive operator.

**isAssignmentOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is an assignment operator.

**isBitwiseOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a bitwise operator.

**isCommaOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is the comma operator.

**isComparisonOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a comparison operator.

**isCompoundAssignmentOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a compound assignment operator.

**isEqualityOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is an equality operator.

**isLogicalOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a logical operator.

**isMultiplicativeOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a multiplicative operator.

**isPtrMemOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a pointer-to-member operator.

**isRelationalOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a relational operator.

**isShiftAssignOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a shift assignment operator.

**isShiftOp** (*BinaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided binary operator is a shift operator.

### 16.8.7 CXXBoolLiteralExpr functions

**getValue** (*CXXBoolLiteralExpr*)  $\Rightarrow$  *Boolean*

Returns the value of the provided Boolean literal expression.

### 16.8.8 CXXCatchStmt functions

**getCaughtType** (*CXXCatchStmt*)  $\Rightarrow$  *ASTType*

Returns the type caught by the provided catch statement of `None` if the provided catch statement does not specify a type, i.e. `catch (...)`.

**getExceptionDecl** (*CXXCatchStmt*)  $\Rightarrow$  *VarDecl*

Returns the variable associated with the provided catch statement or `None` if the provided catch statement does not specify a type, i.e. `catch (...)`.

**getHandlerBlock** (*CXXCatchStmt*)  $\Rightarrow$  *Stmt*

Returns the statement that comprises the handler block of the provided catch statement.

### 16.8.9 CXXConstructExpr functions

**getArg** (*CXXConstructExpr* *Integral*)  $\Rightarrow$  *Expr*

Returns the specified argument in the provided construct expression.

**getConstructor** (*CXXConstructExpr*)  $\Rightarrow$  *CXXConstructorDecl*

Returns the constructor that will be invoked by the provided construct expression.

**getNumArgs** (*CXXConstructExpr*)  $\Rightarrow$  *Integral*

Returns the number of argument in the provided construct expression, including default arguments.

**hadMultipleCandidates** (*CXXConstructExpr*)  $\Rightarrow$  *Boolean*

Returns true if the constructor associated with the provided construct expression was resolved from an overload set with multiple candidates.

**isListInitialization** (*CXXConstructExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided construct expression was written with list-initialization.

**isStdInitListInitialization** (*CXXConstructExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided construct expression was written with list-initialization but was interpreted as forming a `std::initializer_list<T>` from the list and passing that as a single constructor argument.

### 16.8.10 CXXConstructorDecl functions

**getInheritedConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *CXXConstructorDecl*

If the provided constructor is an inheriting constructor, returns the constructor that this constructor is based on, otherwise returns `None`.

**getNumCtorInitializers** (*CXXConstructorDecl*)  $\Rightarrow$  *Integral*

Returns the number of initializers in the provided constructor.

**getTargetConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *CXXConstructorDecl*

If the provided constructor is a delegating constructor, returns the constructor to which it delegates, otherwise returns `None`.

**isConvertingConstructor** (*CXXConstructorDecl* *Boolean*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor can be used for user-defined conversions.

**isCopyConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is a copy constructor.

**isCopyOrMoveConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is a copy constructor or a move constructor.

**isDefaultConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor can be called without any arguments.

**isDelegatingConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is a delegating constructor.

**isExplicit** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is marked as explicit.

**isInheritingConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is an implicit constructor synthesized to model a call to a constructor inherited from a base class.

**isMoveConstructor** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is a move constructor.

**isSpecializationCopyingObject** (*CXXConstructorDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided constructor is a member template specialization that would copy the object to itself.

### 16.8.11 CXXConversionDecl functions

**getConversionType** (*CXXConversionDecl*)  $\Rightarrow$  *ASTType*

Returns the type that the provided conversion function converts to.

**isExplicit** (*CXXConversionDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided conversion function is marked as explicit.

### 16.8.12 CXXDestructorDecl functions

**getOperatorDelete** (*CXXDestructorDecl*)  $\Rightarrow$  *FunctionDecl*

Returns the operator delete function associated with the provided destructor.

**getOperatorDeleteThisArg** (*CXXDestructorDecl*)  $\Rightarrow$  *Expr*

Returns the operator delete 'this' argument associated with the provided destructor.

### 16.8.13 CXXForRangeStmt functions

**getBody** (*CXXForRangeStmt*)  $\Rightarrow$  *Stmt*

Returns the statement that forms the body of the provided for-range statement.

**getInit** (*CXXForRangeStmt*)  $\Rightarrow$  *Stmt*

If the provided for-range statement contains an initialization clause, returns the statement that serves as that clause, otherwise returns *None*.

**getLoopVarStmt** (*CXXForRangeStmt*)  $\Rightarrow$  *DeclStmt*

Returns the declaration statement of the loop variable of the provided for-range statement.

**getLoopVariable** (*CXXForRangeStmt*)  $\Rightarrow$  *VarDecl*

Returns the loop variable of the provided for-range statement.

**getRangeInit** (*CXXForRangeStmt*)  $\Rightarrow$  *Expr*

Returns the range expression of the provided for-range statement.

### 16.8.14 CXXMethodDecl functions

**getOverriddenMethod** (*CXXMethodDecl* *Integral*)  $\Rightarrow$  *CXXMethodDecl*

Returns the specified overridden function of the provided member function.

**getParent** (*CXXMethodDecl*)  $\Rightarrow$  *CXXRecordDecl*

Returns the class in which the provided member function was defined.

**getThisObjectType** (*CXXMethodDecl*)  $\Rightarrow$  *ASTType*

Returns the type pointed to by the 'this' object associated with the provided member function or *None* if the member function is static.

**getThisType** (*CXXMethodDecl*)  $\Rightarrow$  *ASTType*

Returns the type of the 'this' object associated with the provided member function or *None* if the member function is static.

**hasInlineBody** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function has an inlined body.

**isConst** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is const.

**isCopyAssignmentOperator** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is a copy assignment operator.

**isInstance** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is not static.

**isLValueRefQualified** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is lvalue-ref-qualified.

**isLambdaStaticInvoker** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is a lambda closure type's static member function that is used for the result of the lambda's conversion to function pointer.

**isMoveAssignmentOperator** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is a move assignment operator.

**isRValueRefQualified** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is rvalue-ref-qualified.

**isRefQualified** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided member function is ref-qualified.

**isStatic** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*

Returns true if provided member function is static.  
**isVirtual** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided member function is virtual.  
**isVolatile** (*CXXMethodDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided member function is volatile.  
**numOverriddenMethods** (*CXXMethodDecl*)  $\Rightarrow$  *Integral*  
 Returns the number of member functions overridden by the provided member function.

### 16.8.15 CXXRecordDecl functions

**defaultedCopyConstructorIsDeleted** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if a defaulted copy constructor for the provided class would be deleted.  
**defaultedDestructorIsDeleted** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if a defaulted destructor for the provided class would be deleted.  
**defaultedMoveConstructorIsDeleted** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if a defaulted move constructor for the provided class would be deleted.  
**getBase** (*CXXRecordDecl* *Integral*)  $\Rightarrow$  *ASTType*  
 Returns the specified base of the provided class.  
**getConstructor** (*CXXRecordDecl* *Integral*)  $\Rightarrow$  *CXXConstructorDecl*  
 Returns the specified constructor of the provided class.  
**getDependentLambdaCallOperator** (*CXXRecordDecl*)  $\Rightarrow$  *FunctionTemplateDecl*  
 If the provided class corresponds to a templated closure type, returns the dependent lambda call operator of the closure type, otherwise returns *None*.  
**getDescribedClassTemplate** (*CXXRecordDecl*)  $\Rightarrow$  *ClassTemplateDecl*  
 Returns the class template that is described the the provided class if any, otherwise returns *None*.  
**getDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *CXXDestructorDecl*  
 Returns the destructor of the provided class.  
**getFriend** (*CXXRecordDecl* *Integral*)  $\Rightarrow$  *FriendDecl*  
 Returns the specified friend of the provided class.  
**getInstantiatedFromMemberClass** (*CXXRecordDecl*)  $\Rightarrow$  *CXXRecordDecl*  
 If the provided class is a member class instantiation, returns the member class from which it was instantiated, otherwise returns *None*.  
**getLambdaCallOperator** (*CXXRecordDecl*)  $\Rightarrow$  *CXXMethodDecl*  
 If the provided class is a closure type, returns the lambda call operator, otherwise returns *None*.  
**getLambdaStaticInvoker** (*CXXRecordDecl*)  $\Rightarrow$  *CXXMethodDecl*  
 If the provided class corresponds to a lambda, returns the lambda static invoker, otherwise returns *None*.  
**getMethod** (*CXXRecordDecl* *Integral*)  $\Rightarrow$  *CXXMethodDecl*  
 Returns the specified member function of the provided class.  
**getNumBases** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*  
 Returns the number of bases the provided class was defined with or *None* if the class is not defined.  
**getNumVBases** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*  
 Returns the number of virtual bases the provided class was defined with or *None* if the class is not defined.  
**getTemplateInstantiationPattern** (*CXXRecordDecl*)  $\Rightarrow$  *CXXRecordDecl*  
 If the provided class is a template instantiation, returns the class from which it could have been instantiated, otherwise returns *None*.  
**getVirtualBase** (*CXXRecordDecl* *Integral*)  $\Rightarrow$  *ASTType*  
 Returns the specified virtual base of the provided class.  
**hasConstexprDefaultConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided class as a constexpr default constructor.  
**hasConstexprDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided class has a constexpr destructor.  
**hasConstexprNonCopyMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided class has a constexpr constructor that is neither a copy constructor nor a move constructor.  
**hasCopyAssignmentWithConstParam** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided class has a copy assignment operator with reference to const-qualified type as a parameter.

**hasCopyConstructorWithConstParam** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a copy constructor with reference to const-qualified type as a parameter.

**hasDefaultConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has one or more default constructors.

**hasDefinition** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the class associated with the provided declaration is defined in the current module.

**hasDirectFields** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class declared any non-static data members.

**hasFriends** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class was defined with one or more friends.

**hasInClassInitializer** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has an in-class initializer for any non-static data members.

**hasInheritedAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a using-declaration that names a base class assignment operator.

**hasInheritedConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a using-declaration that names a user-declared base class constructor.

**hasIrrelevantDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a destructor with no semantic effect.

**hasMoveAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a move assignment operator.

**hasMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a move constructor.

**hasMutableFields** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class contains any mutable fields or inherits from a class that does.

**hasNonLiteralTypeFieldsOrBases** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has non-literal type, non-literal fields, or non-literal bases.

**hasNonTrivialCopyAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial copy assignment operator.

**hasNonTrivialCopyConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial copy constructor.

**hasNonTrivialDefaultConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial default constructor.

**hasNonTrivialDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial destructor.

**hasNonTrivialMoveAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial move assignment operator.

**hasNonTrivialMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a non-trivial move constructor.

**hasPrivateFields** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class declares any private non-static data members.

**hasProtectedFields** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class declares any protected non-static data members.

**hasSimpleCopyAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a single, accessible, unambiguous copy assignment operator that is not deleted.

**hasSimpleCopyConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a single, accessible, unambiguous copy constructor that is not deleted.

**hasSimpleDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has an accessible destructor that is not deleted.

**hasSimpleMoveAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a single, accessible, unambiguous move assignment operator that is not deleted.

**hasSimpleMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a single, accessible, unambiguous move constructor that is not deleted.

**hasTrivialCopyAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has a trivial copy assignment operator.

**hasTrivialCopyConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a trivial copy constructor.

**hasTrivialDefaultConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a trivial default constructor.

**hasTrivialDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a trivial destructor.

**hasTrivialMoveAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a trivial move assignment operator.

**hasTrivialMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a trivial move constructor.

**hasUninitializedReferenceMember** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class contains one or more reference members and the class does not have a user-declared constructor.

**hasUserDeclaredConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared constructor.

**hasUserDeclaredCopyAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared copy assignment operator.

**hasUserDeclaredCopyConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared copy constructor.

**hasUserDeclaredDestructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared destructor.

**hasUserDeclaredMoveAssignment** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared move assignment operator.

**hasUserDeclaredMoveConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-declared move constructor.

**hasUserDeclaredMoveOperation** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has user-declared move constructor or a user-declared move assignment operator.

**hasUserProvidedDefaultConstructor** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has a user-provided default constructor.

**hasVariantMembers** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has any variant members.

**isAbstract** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class declares a pure virtual function or inherits a pure virtual function that it does not override.

**isAggregate** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class is an aggregate. An aggregate is a class having no user-declared constructors and no base classes, virtual functions, or non-public non-static data members.

**isAnyDestructorNoReturn** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class' destructor, or any implicitly invoked destructor, are declared as not returning.

**isCLike** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class does not utilize C++ -specific features such as non-public fields, base classes, member functions, use of the 'class' keyword, etc.

**isCXX11StandardLayout** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class has standard layout as defined by C++ 11.

**isDerivedFrom** (*CXXRecordDecl* *CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the first provided class is derived from the second provided class.

**isEmpty** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class would be considered empty by the `std::is_empty` type trait. A class is considered empty if it is a non-union class which does not have any virtual functions or virtual base classes, whose only non-static data members are zero width bit-fields, and any base classes are also empty.

**isGenericLambda** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided class describes a generic lambda function object.

**isInterfaceLike** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is declared using the non-standard 'interface' keyword or has a single interface-like base class and no user-declared constructors, destructors, conversion functions, friends, fields, virtual bases, member function definitions.

**isLambda** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class describes a lambda function object.

**isLiteral** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is a literal type in the current C++ language mode.

**isLocalClass** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

If the provided class is a local class, returns the enclosing function declaration, otherwise returns *None*.

**isPOD** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is a POD type as defined in C++ TR1.

**isPolymorphic** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class contains or inherits any virtual functions.

**isProvablyNotDerivedFrom** (*CXXRecordDecl* *CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the first provided class is provably not derived from the second provided class.

**isStandardLayout** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class has standard layout as defined by C++ .

**isStructural** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is a structural type.

**isTrivial** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is trivial, that is it has a trivial default constructor and is trivially copyable.

**isTriviallyCopyable** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is trivially copyable.

**isVirtuallyDerivedFrom** (*CXXRecordDecl* *CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the first provided class is virtually derived from the second provided class.

**lambdaIsDefaultConstructibleAndAssignable** (*CXXRecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided class is a closure type with implicit default constructor and copy and move assignment operators.

**numBases** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of bases of the provided class.

**numConstructors** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of constructors of the provided class.

**numFriends** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of friends of the provided class.

**numMethods** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of member functions of the provided class.

**numVirtualBases** (*CXXRecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of virtual bases of the provided class.

### 16.8.16 CXXTryStmt functions

**getHandler** (*CXXTryStmt* *Integral*)  $\Rightarrow$  *CXXCatchStmt*

Returns the 'catch' statement corresponding to the specified exception handler for the provided 'try' statement.

**getNumHandlers** (*CXXTryStmt*)  $\Rightarrow$  *Integral*

Returns the number of exception handlers for the provided 'try' statement.

**getTryBlock** (*CXXTryStmt*)  $\Rightarrow$  *CompoundStmt*

Returns the compound statement forming the try-block of the provided 'try' statement.

### 16.8.17 CallExpr functions

**getArg** (*CallExpr* *Integral*)  $\Rightarrow$  *Expr*

Returns the specified argument in the provided call expression.

**getCallReturnType** (*CallExpr*)  $\Rightarrow$  *ASTType*

Returns the *return type* of the provided call expression which may be different than the *type* of the expression for functions returning a reference type.



**getCallee** (*CallExpr*)  $\Rightarrow$  *Expr*

Returns the expression that comprises the function being called in the provided call expression.

**getCalleeDecl** (*CallExpr*)  $\Rightarrow$  *Decl*

Returns the declaration associated with the callee in the provided call expression.

**getDirectCallee** (*CallExpr*)  $\Rightarrow$  *FunctionDecl*

Returns the declaration of the function called by the provided call expression if this is a direct function call (e.g. not a call through a function pointer variable).

**getNumArgs** (*CallExpr*)  $\Rightarrow$  *Integral*

Returns the number of arguments in the call to the provided call expression, including any default arguments.

**hasUnusedResultAttr** (*CallExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided call expression involves a return type or a called function declared with the `nodiscard` attribute.

**isCallToStdMove** (*CallExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided call expression represents a call to `std::move`.

**usesADL** (*CallExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided call expression employs argument-dependent lookup.

### 16.8.18 CastExpr functions

**getCastKindName** (*CastExpr*)  $\Rightarrow$  *String*

Returns a string representing the kind of conversion employed by the provided conversion expression.

**getConversionFunction** (*CastExpr*)  $\Rightarrow$  *NamedDecl*

Returns the conversion function used to perform the provided conversion if any, otherwise returns `None`.

**getSubExpr** (*CastExpr*)  $\Rightarrow$  *Expr*

Returns the converted expression of the provided conversion expression.

### 16.8.19 CharacterLiteral functions

**getValue** (*CharacterLiteral*)  $\Rightarrow$  *Integral*

Returns the numeric value of the provided character literal.

**isAscii** (*CharacterLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided character literal is an ascii character.

**isUTF16** (*CharacterLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided character literal is a UTF16 character.

**isUTF32** (*CharacterLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided character literal is a UTF32 character.

**isUTF8** (*CharacterLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided character literal is a UTF8 character.

**isWide** (*CharacterLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided character literal is a wide character.

### 16.8.20 CompoundStmt functions

**getNumStmts** (*CompoundStmt*)  $\Rightarrow$  *Integral*

Returns the number of (top-level) statements in the provided compound statement.

**getStmt** (*CompoundStmt* *Integral*)  $\Rightarrow$  *Stmt*

Returns the specified statement in the provided compound statement.

**getStmtExprResult** (*CompoundStmt*)  $\Rightarrow$  *Stmt*

Returns the statement that would be used as the result of a GCC compound expression.

### 16.8.21 CoreturnStmt functions

**getOperand** (*CoreturnStmt*)  $\Rightarrow$  *Expr*

Returns the return value expression of the provided coreturn statement if any, otherwise returns `None`.

**getPromiseCall** (*CoreturnStmt*)  $\Rightarrow$  *Expr*

Returns the promise call that results from the provided coreturn statement if any, otherwise returns `None`.  
**isImplicit** (*CoreturnStmt*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided coreturn statement is implicit.

### 16.8.22 CoroutineBodyStmt functions

**getAllocate** (*CoroutineBodyStmt*)  $\Rightarrow$  *Expr*  
 Returns the allocation expression for the provided coroutine.  
**getBody** (*CoroutineBodyStmt*)  $\Rightarrow$  *Stmt*  
 Returns the body of the provided coroutine.  
**getDeallocate** (*CoroutineBodyStmt*)  $\Rightarrow$  *Expr*  
 Returns the deallocation expression for the provided coroutine.

### 16.8.23 Decl functions

**dump** (*Decl*)  $\Rightarrow$  *String*  
 Returns a string containing the AST tree of the provided declaration.  
**getAccess** (*Decl*)  $\Rightarrow$  *String*  
 Returns a string ("public", "protected", "private", or "none") indicating the access of the provided declaration.  
**getBeginLoc** (*Decl*)  $\Rightarrow$  *Location*  
 Returns the Location associated with the beginning of the provided declaration.  
**getCanonicalDecl** (*Decl*)  $\Rightarrow$  *Decl*  
 Returns the canonical declaration of the provided declaration.  
**getDeclContext** (*Decl*)  $\Rightarrow$  *Decl*  
 Returns the semantic declaration context of the provided declaration.  
**getDeclKindName** (*Decl*)  $\Rightarrow$  *String*  
 Returns a string representing the kind of the provided declaration.  
**getEndLoc** (*Decl*)  $\Rightarrow$  *Location*  
 Returns the Location associated with the end of the provided declaration.  
**getLexicalDeclContext** (*Decl*)  $\Rightarrow$  *Decl*  
 Returns the lexical declaration context of the provided declaration.  
**getMaxAlignment** (*Decl*)  $\Rightarrow$  *String*  
 Returns the maximum alignment as specified by attributes applied to the provided declaration or 0.  
**getParentFunctionOrMethod** (*Decl*)  $\Rightarrow$  *FunctionDecl*  
 If the provided declaration was declared within a function, returns that function, otherwise returns `None`.  
**getTemplateDepth** (*Decl*)  $\Rightarrow$  *Integral*  
 Returns the number of levels of template parameter surrounding the provided declaration.  
**hasBody** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration has a body.  
**isCanonicalDecl** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns the canonical declaration associated with the provided declaration.  
**isFirstDecl** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration is the first declaration of the declared entity.  
**isFunctionOrFunctionTemplate** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration is a function or function template.  
**isImplicit** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration was implicitly generated.  
**isInAnonymousNamespace** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration was declared in an anonymous namespace.  
**isInStdNmespace** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration was declared in the std namespace.  
**isInvalidDecl** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration is invalid, e.g. if semantic error occurred during processing of the declaration.  
**isLocalExternDecl** (*Decl*)  $\Rightarrow$  *Boolean*  
 Returns true if the provided declaration is a block-scope declaration with linkage.

**isOutOfLine** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration was declared outside of its semantic context.

**isParameterPack** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration is a parameter pack.

**isReferenced** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration, or any of its redeclarations, was referenced.

**isTemplateDecl** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration is a template.

**isTemplateParameter** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration is a template parameter.

**isTemplateParameterPack** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration is a template parameter pack.

**isTemplated** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration is a template, template pattern, or is within a dependent context.

**isUsed** (*Decl*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration, or any of its redeclarations, was used in a manner that requires a definition.

### 16.8.24 DeclRefExpr functions

**getDecl** (*DeclRefExpr*)  $\Rightarrow$  *ValueDecl*

Returns the declaration to which the provided declaration reference expression refers.

**getFoundDecl** (*DeclRefExpr*)  $\Rightarrow$  *NamedDecl*

Returns the named declaration through which the provided declaration reference expression occurs which may be different than the value returned by **getDecl** such as in the presence of using declarations.

**hasExplicitTemplateArgs** (*DeclRefExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration reference expression was followed by an explicit template argument list.

**hasTemplateKeyword** (*DeclRefExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided declaration reference expression was preceded by the **template** keyword.

### 16.8.25 DoStmt functions

**getBody** (*DoStmt*)  $\Rightarrow$  *Stmt*

Returns the body of the provided 'do' statement.

**getCond** (*DoStmt*)  $\Rightarrow$  *Expr*

Returns the condition expression of the provided 'do' statement.

### 16.8.26 EnumConstantDecl functions

**getInitExpr** (*EnumConstantDecl*)  $\Rightarrow$  *Expr*

Returns the expression used to specify the value of the provided enumeration constant or **None**.

**getInitVal** (*EnumConstantDecl*)  $\Rightarrow$  *Integral*

Returns the value of the provided enumeration constant.

### 16.8.27 EnumDecl functions

**getEnumerator** (*EnumDecl* *Integral*)  $\Rightarrow$  *EnumConstantDecl*

Returns the specified enumerator of the provided enumeration.

**getIntegerType** (*EnumDecl*)  $\Rightarrow$  *ASTType*

Returns the underlying integer type of the provided enumeration.

**getNumEnumerators** (*EnumDecl*)  $\Rightarrow$  *Integral*

Returns the number of enumerators defined for the provided enumeration.

**getNumNegativeBits** (*EnumDecl*)  $\Rightarrow$  *Integral*

Returns the number of bits needed to represent the smallest negative enumerator for the provided enumeration.

**getNumPositiveBits** (*EnumDecl*)  $\Rightarrow$  *Integral*

Returns the number of bits needed to represent the largest non-negative enumerator for the provided enumeration.

**getPromotionType** (*EnumDecl*)  $\Rightarrow$  *ASTType*

Returns the type that enumerators of the provided enumeration will be promoted to.

**isComplete** (*EnumDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided enumeration is considered to be a complete type.

**isFixed** (*EnumDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided enumeration was declared with a fixed underlying type using the C++ 11 or a compiler-specific mechanism.

**isScoped** (*EnumDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided enumeration is a C++ 11 scoped enumeration.

**isScopedUsingClassTag** (*EnumDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided enumeration is a scoped enumeration defined using the 'class' keyword.

### 16.8.28 Expr functions

**containsUnexpandedParameterPack** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression contains an unexpanded parameter pack.

**evaluateAsBooleanCondition** (*Expr*)  $\Rightarrow$  *Boolean*

If the provided expression is a constant expression which can be manipulated to a Boolean condition, returns the value of that manipulation, otherwise returns *None*.

**evaluateAsFloat** (*Expr*)  $\Rightarrow$  *Floating*

If the provided expression is a constant expression that can be manipulated to an integer, returns the value of that manipulation, otherwise returns *None*.

**evaluateAsInt** (*Expr*)  $\Rightarrow$  *Integral*

If the provided expression is a constant expression that can be manipulated to an integer, returns the value of that manipulation, otherwise returns *None*.

**getExprLoc** (*Expr*)  $\Rightarrow$  *Location*

Returns the preferred diagnostic location of the provided expression, typically the location of the operator itself as opposed to an operand.

**getIntegerConstantExpr** (*Expr*)  $\Rightarrow$  *Integral*

If the provided expression is an integer constant expression, returns the value of the expression, otherwise returns *None*.

**getSourceBitField** (*Expr*)  $\Rightarrow$  *FieldDecl*

If the provided expression refers to a bit-field, returns the declaration of that bit-field, otherwise returns *None*.

**getType** (*Expr*)  $\Rightarrow$  *ASTType*

Returns the type of the provided expression.

**hasNonTrivialCall** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression involves a call to a non-trivial function.

**ignoreCasts** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any explicit casts starting at the provided expression.

**ignoreImpCasts** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any implicit casts starting at the provided expression.

**ignoreImplicit** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over the same nodes that **ignoreImpCasts** does as well as any **MaterializeTemporaryExpr** and **CXXBindTemporaryExpr** nodes at the provided expression.

**ignoreImplicitAsWritten** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over the same nodes that **ignoreImplicit** does as well as any implicit calls to constructors and conversion functions at the provided expression.

**ignoreParenBaseCasts** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any parentheses and derived-to-base casts starting at the provided expression.

**ignoreParenCasts** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any parentheses and/or implicit/explicit casts starting at the provided expression.

**ignoreParenImpCasts** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any parentheses and/or implicit casts starting at the provided expression.

**ignoreParens** (*Expr*)  $\Rightarrow$  *Expr*

Returns the expression resulting from skipping over any parentheses starting at the provided expression.

**isBoundMemberFunction** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a bound member function.

**isCXX11ConstantExpr** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a constant expression in C++ 11.

**isCXX98IntegralConstantExpr** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is an integer constant expression in C++ 98.

**isConstantExpr** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a constant expression.

**isDefaultArgument** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a default function argument.

**isGLValue** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a *gl-value* according to the current language mode.

**isImplicitCXXThis** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is an implicit reference to the C++ **this** pointer.

**isIntegerConstantExpr** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true the provided expression is an integer constant expression.

**isKnownToHaveBooleanValue** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is an integer constant expression known to have a value of 0 or 1.

**isLValue** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a *l-value* according to the current language mode.

**isNullPointerConstant** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression reduces to a null pointer constant.

**isRValue** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a *r-value* in C mode or *pr-value* in C++ mode.

**isTypeDependent** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the type of the provided expression depends on a template parameter.

**isValueDependent** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the value of the provided expression depends on a template parameter.

**isXValue** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a *xl-value* according to the current language mode.

**refersToBitField** (*Expr*)  $\Rightarrow$  *Boolean*

Returns true if the provided expression is a *gl-value* that potentially refers to a bit-field.

### 16.8.29 FieldDecl functions

**getBitWidthValue** (*FieldDecl*)  $\Rightarrow$  *Integral*

If the provided field is a bit-field, returns the bit width of the bit-field, otherwise returns **None**.

**getInClassInitStyle** (*FieldDecl*)  $\Rightarrow$  *String*

Returns a string (one of "None", "Copy", or "List") representing the initialization style used to initialize the provided field.

**getInClassInitializer** (*FieldDecl*)  $\Rightarrow$  *Expr*

If the provided field has an in-class initializer, returns the expression used in that initializer, otherwise returns **None**.

**getParent** (*FieldDecl*)  $\Rightarrow$  *RecordDecl*

Returns the parent class of the provided field.

**hasInClassInitializer** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field has an in-class initializer.

**isAnonymousStructOrUnion** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field is an anonymous struct or union.

**isBitField** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field is a bit-field.

**isMutable** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field was defined with the 'mutable' keyword.

**isUnnamedBitfield** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field is an unnamed bit-field.

**isZeroLengthBitField** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field is a zero-width bit-field.

**isZeroSize** (*FieldDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided field is a zero-width bit-field or is of empty class type.

### 16.8.30 FloatingLiteral functions

**getValue** (*FloatingLiteral*)  $\Rightarrow$  *Floating*

Returns the value of the provided floating point literal.

**isExact** (*FloatingLiteral*)  $\Rightarrow$  *Boolean*

Returns true if the provided floating point literal can be exactly represented in its corresponding type.

### 16.8.31 ForStmt functions

**getBody** (*ForStmt*)  $\Rightarrow$  *Stmt*

Returns the statement that forms the body of the provided **for** statement.

**getCond** (*ForStmt*)  $\Rightarrow$  *Expr*

Returns the statement that comprises the second/condition clause of the provided **for** statement if any, otherwise returns *None*.

**getConditionVariable** (*ForStmt*)  $\Rightarrow$  *VarDecl*

Returns the condition variable declared in the second clause of the provided **for** statement if any, otherwise returns *None*.

**getConditionVariableDeclStmt** (*ForStmt*)  $\Rightarrow$  *DeclStmt*

Returns the declaration statement of the condition variable declared in the second clause of the provided **for** statement if any, otherwise returns *None*.

**getInc** (*ForStmt*)  $\Rightarrow$  *Expr*

Returns the statement that comprises the third/increment clause of the provided **for** statement if any, otherwise returns *None*.

**getInit** (*ForStmt*)  $\Rightarrow$  *Stmt*

Returns the statement that comprises the first/initialization clause of the provided **for** statement if any, otherwise returns *None*.

### 16.8.32 FriendDecl functions

**getFriendDecl** (*FriendDecl*)  $\Rightarrow$  *NamedDecl*

If the provided friend declaration does not name a type, returns the inner declaration, otherwise *None* is returned.

**getFriendType** (*FriendDecl*)  $\Rightarrow$  *ASTType*

If the provided friend declaration names a type, that type is returned, otherwise *None* is returned.

### 16.8.33 FriendTemplateDecl functions

**getFriendDecl** (*FriendTemplateDecl*)  $\Rightarrow$  *NamedDecl*

If the provided friend declaration names a templated function, that function is returned, otherwise *None* is returned.

**getFriendType** (*FriendTemplateDecl*)  $\Rightarrow$  *ASTType*

If the provided friend declaration names a templated type, that type is returned, otherwise *None* is returned.

### 16.8.34 FunctionDecl functions

**doesThisDeclarationHaveABody** (*FunctionDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided function declaration has a body.

**getBody** (*FunctionDecl*)  $\Rightarrow$  *Stmt*

Returns the body associated with the provided function or *None* if there is no body.

**getCallResultType** (*FunctionDecl*)  $\Rightarrow$  *ASTType*

Returns the type of an expression that calls the provided function.

**getDeclaredReturnType** (*FunctionDecl*)  $\Rightarrow$  *ASTType*

- Returns the type that the provided function is declared as returning.
- getDefinition** (*FunctionDecl*)  $\Rightarrow$  *FunctionDecl*
- Returns the declaration of the provided function that contains a body if any, otherwise returns *None*.
- getDescribedFunctionTemplate** (*FunctionDecl*)  $\Rightarrow$  *FunctionTemplateDecl*
- Returns the function template that is described by the provided function or *None*.
- getInstantiatedFromMemberFunction** (*FunctionDecl*)  $\Rightarrow$  *FunctionDecl*
- If the provided function is an instantiation of a member function of a class template specialization, returns the function from which it was instantiated, otherwise returns *None*.
- getMinRequiredArguments** (*FunctionDecl*)  $\Rightarrow$  *Integral*
- Returns the minimum number of arguments needed to invoke the provided function.
- getNumParams** (*FunctionDecl*)  $\Rightarrow$  *Integral*
- Returns the number of parameters that the provided function was declared as receiving.
- getParamDecl** (*FunctionDecl* *Integral*)  $\Rightarrow$  *ParmVarDecl*
- Returns the specified parameter declaration of the provided function.
- getPointOfInstantiation** (*FunctionDecl*)  $\Rightarrow$  *Location*
- Returns the location in which the provided function was instantiated or *None* if it is not an instantiation.
- getPrimaryTemplate** (*FunctionDecl*)  $\Rightarrow$  *FunctionTemplateDecl*
- Returns the function template that the provided function specializes or was instantiated from if any, otherwise returns *None*.
- getReturnType** (*FunctionDecl*)  $\Rightarrow$  *ASTType*
- Returns the type that the provided function returns.
- getStorageClass** (*FunctionDecl*)  $\Rightarrow$  *String*
- Returns a string (one of "None", "Extern", "Static", "PrivateExtern", "Auto", or "Register") representing the storage class of the provided function.
- getTemplateInstantiationPattern** (*FunctionDecl*)  $\Rightarrow$  *FunctionDecl*
- If the provided function is an instantiation, returns the function from which it could have been instantiated, otherwise returns *None*.
- hasImplicitReturnZero** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true falling off the end of the provided function implicitly returns zero.
- hasInheritedPrototype** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function has a prototype inherited from a previous declaration.
- hasOneParamOrDefaultArgs** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function may be invoked with a single argument.
- hasPrototype** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function has either a written prototype or a prototype obtained from merging the function declaration with one that does.
- hasSkippedBody** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the body of the provided function was skipped such as with the `-skip_function` option.
- hasTrivialBody** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the body associated with the provided function is trivial.
- hasWrittenPrototype** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function has a written prototype.
- isConsteval** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function is a consteval function.
- isConstexpr** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function is constexpr.
- isConstexprSpecified** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function was specified as constexpr.
- isDefaulted** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function is defaulted.
- isDefined** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if there is a definition associated with the provided function.
- isDeleted** (*FunctionDecl*)  $\Rightarrow$  *Boolean*
- Returns true if the provided function is deleted.

- isDeletedAsWritten** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function was marked as deleted in the source code.
- isDestroyingOperatorDelete** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a destroying operator delete.
- isExplicitlyDefaulted** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is explicitly defaulted.
- isExternC** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function has external C linkage.
- isFunctionTemplateSpecialization** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a function template specialization.
- isGlobal** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a global function.
- isImplicitlyInstantiable** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a function template specialization or a member of a class template specialization that can be implicitly instantiated.
- isInExternCContext** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is subject to a C++ extern "C" linkage specifier.
- isInExternCXXContext** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is subject to a C++ extern "C++ " linkage specifier.
- isInlineSpecified** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function was declared using the `inline` keyword.
- isInlined** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is an inline function, e.g. it was declared using `inline`, `constexpr`, or is a member function defined in the body of its class.
- isLateTemplateParsed** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a templated function that will be late parsed.
- isMain** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is the main function.
- isNoReturn** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function was declared as not returning.
- isOverloadedOperator** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is an overloaded operator.
- isPure** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a pure virtual function.
- isReplaceableGlobalAllocationFunction** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is one of the replaceable global allocation functions.
- isReservedGlobalPlacementOperator** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is one of the reserved global placement operators.
- isStatic** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a static function.
- isTemplateInstantiation** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function was instantiated from a function template.
- isThisDeclarationADefinition** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function declaration is a definition.
- isThisDeclarationInstantiatedFromAFriendDefinition** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function declaration is instantiated from a friend definition.
- isTrivial** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is a trivial special member function.
- isUserProvided** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is used-declared and not deleted or defaulted in its first declaration.
- isVariadic** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function is variadic.
- isVirtualAsWritten** (*FunctionDecl*)  $\Rightarrow$  *Boolean*  
Returns true if the provided function was explicitly declared virtual.



**16.8.35 GotoStmt functions****getLabel** (*GotoStmt*)  $\Rightarrow$  *LabelDecl*

Returns the label statement corresponding to the provided 'goto' statement.

**16.8.36 IfStmt functions****getCond** (*IfStmt*)  $\Rightarrow$  *Expr*

Returns the condition expression of the provided if statement.

**getConditionVariable** (*IfStmt*)  $\Rightarrow$  *VarDecl*Returns the condition variable declared in the provided if statement if any, otherwise returns *None*.**getConditionVariableDeclStmt** (*IfStmt*)  $\Rightarrow$  *DeclStmt*If the provided if statement declares a condition variable, returns the declaration statement associated with the condition variable, otherwise returns *None*.**getElse** (*IfStmt*)  $\Rightarrow$  *Stmt*Returns the statement forming the *else* clause of the provided if statement or *None* if there is no *else* clause.**getInit** (*IfStmt*)  $\Rightarrow$  *Stmt*If the provided if statement contains an initialization clause, returns the statement that serves as that clause, otherwise returns *None*.**getNondiscardedCase** (*IfStmt*)  $\Rightarrow$  *Stmt*If the provided if statement is an if *constexpr* statement, returns the substatement that will be evaluated, otherwise returns *None*.**getThen** (*IfStmt*)  $\Rightarrow$  *Stmt*Returns the statement forming the *then* clause of the provided if statement.**isConstexpr** (*IfStmt*)  $\Rightarrow$  *Boolean*Returns true if the provided if statement is an if *constexpr* statement.**16.8.37 IntegerLiteral functions****getValue** (*IntegerLiteral*)  $\Rightarrow$  *Integral*

Returns the value of the provided integer literal.

**16.8.38 LabelDecl functions****getStmt** (*LabelDecl*)  $\Rightarrow$  *LabelStmt*

Returns the substatement of the provided label.

**isGnuLocal** (*LabelDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided label is a GNU local label.

**16.8.39 LabelStmt functions****getDecl** (*LabelStmt*)  $\Rightarrow$  *LabelDecl*

Returns the declaration of the label referenced by the provided label statement.

**getName** (*LabelStmt*)  $\Rightarrow$  *String*

Returns the name of the label referenced by the provided label statement.

**getSubStmt** (*LabelStmt*)  $\Rightarrow$  *Stmt*

Returns the substatement of the provided label statement.

**16.8.40 LambdaExpr functions****getCallOperator** (*LambdaExpr*)  $\Rightarrow$  *CXXMethodDecl*

Returns the function call operator associated with the provided lambda.

**getDependentCallOperator** (*LambdaExpr*)  $\Rightarrow$  *FunctionTemplateDecl*Returns the function template call operator associated with the provided lambda if any, otherwise returns *None*.**getLambdaClass** (*LambdaExpr*)  $\Rightarrow$  *CXXRecordDecl*

Returns the class that corresponds to the provided lambda.

**hasDefaultCapture** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda has a capture-default.

**hasDefaultCaptureByCopy** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda has a capture-default of capture-by-copy.

**hasDefaultCaptureByReference** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda has a capture-default of capture-by-reference.

**hasExplicitParameters** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda is declared with an explicit parameter list.

**hasExplicitResultType** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the result type was explicitly specified for the provided lambda.

**isGenericLambda** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda is a generic lambda.

**isMutable** (*LambdaExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided lambda captures any values that may be modified in the lambda.

#### 16.8.41 LinkageSpecDecl functions

**hasBraces** (*LinkageSpecDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided linkage declaration was defined with braces.

**isCLanguageLinkage** (*LinkageSpecDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided linkage declaration is a C-language linkage declaration.

**isCXXLanguageLinkage** (*LinkageSpecDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided linkage declaration is a C++ -language linkage declaration.

#### 16.8.42 Location functions

**getBufferName** (*Location*)  $\Rightarrow$  *String*

Returns the buffer name of the provided Location.

**getSpellingColumnNumber** (*Location*)  $\Rightarrow$  *Integral*

Returns the column number of the provided Location.

**getSpellingLineNumber** (*Location*)  $\Rightarrow$  *Integral*

Returns the line number of the provided Location.

**isLibraryRegion** (*Location*)  $\Rightarrow$  *Boolean*

Returns true if the provided location is in a library region.

**isMacroLocation** (*Location*)  $\Rightarrow$  *Boolean*

Returns true if the provided location is the result of macro expansion.

**isMainFile** (*Location*)  $\Rightarrow$  *Boolean*

Returns true if the provided location resides in the main source file of the current translation unit.

#### 16.8.43 MemberExpr functions

**getBase** (*MemberExpr*)  $\Rightarrow$  *Expr*

Returns the expression forming the base (LHS operand) of the provided member expression.

**getMemberDecl** (*MemberExpr*)  $\Rightarrow$  *ValueDecl*

Returns the member declaration corresponding to the provided member expression.

**hadMultipleCandidates** (*MemberExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided member expression refers to a function that was resolved from an overloaded set having multiple candidates.

**hasQualifier** (*MemberExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided member expression used a C++ nested-name specifier to refer to the member, e.g. `x->Base::foo`.

**hasTemplateKeyword** (*MemberExpr*)  $\Rightarrow$  *Boolean*

Returns true if the member name in the provided member expression was preceded by the `template` keyword.

**isArrow** (*MemberExpr*)  $\Rightarrow$  *Boolean*

Returns true if the provided member expression used the arrow syntax, e.g. `x->foo` as opposed to `x.foo`.

**performsVirtualDispatch** (*MemberExpr*)  $\Rightarrow$  *Boolean*

Returns true if virtual dispatch is performed for the provided member expression.

#### 16.8.44 NamedDecl functions

**getNameAsString** (*NamedDecl*)  $\Rightarrow$  *String*

Returns a string containing the name of the provided named declaration or "(unnamed)" if the provided declaration does not have a name.

**getQualifiedNameAsString** (*NamedDecl*)  $\Rightarrow$  *String*

Returns a string containing the fully qualified name of the provided named declaration.

**getUnderlyingDecl** (*NamedDecl*)  $\Rightarrow$  *NamedDecl*

Returns the underlying declaration of the provided named declaration looking through any **using** declarations.

**hasExternalFormalLinkage** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration has external linkage.

**hasLinkage** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration has linkage.

**hasName** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration has a name.

**isCXXClassMember** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration is a C++ class member.

**isCXXInstanceMember** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration is a C++ class instance member.

**isExternallyDeclarable** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration can be redeclared in a different translation unit.

**isExternallyVisible** (*NamedDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided named declaration is externally visible.

#### 16.8.45 NamespaceAliasDecl functions

**getCanonicalDecl** (*NamespaceAliasDecl*)  $\Rightarrow$  *NamespaceAliasDecl*

Returns the canonical declaration of the provided namespace alias declaration.

**getNamespace** (*NamespaceAliasDecl*)  $\Rightarrow$  *NamespaceDecl*

Returns the namespace declaration corresponding to the provided namespace alias declaration.

#### 16.8.46 NamespaceDecl functions

**getAnonymousNamespace** (*NamespaceDecl*)  $\Rightarrow$  *NamespaceDecl*

Returns the anonymous namespace nested within the provided namespace if any, otherwise returns **None**.

**getOriginalNamespace** (*NamespaceDecl*)  $\Rightarrow$  *NamespaceDecl*

Returns the first declaration of the provided namespace.

**isAnonymousNamespace** (*NamespaceDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided namespace is an anonymous namespace.

**isInline** (*NamespaceDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided namespace is an inline namespace.

**isOriginalNamespace** (*NamespaceDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided namespace is the first declaration of the namespace.

#### 16.8.47 NullStmt functions

**hasLeadingEmptyMacro** (*NullStmt*)  $\Rightarrow$  *Boolean*

Returns true if the provided null statement follows a macro that expanded to nothing.

#### 16.8.48 ParenExpr functions

**getSubExpr** (*ParenExpr*)  $\Rightarrow$  *Expr*

Returns the enclosed expression of the provided parenthesized expression.

### 16.8.49 ParmVarDecl functions

**getDefaultArg** (*ParmVarDecl*)  $\Rightarrow$  *Expr*

If the provided parameter was declared with a default argument, returns the expression comprising that argument, otherwise returns *None*.

**getFunctionScopeIndex** (*ParmVarDecl*)  $\Rightarrow$  *Integral*

Returns the index of the provided parameter within its prototype.

**getOriginalType** (*ParmVarDecl*)  $\Rightarrow$  *ASTType*

Returns the original type of the provided parameter before e.g. array to pointer decay.

**hasDefaultArg** (*ParmVarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided parameter was declared with a default argument.

**hasInheritedDefaultArg** (*ParmVarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided parameter inherits a default argument from an overridden function.

**isKNRPromoted** (*ParmVarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided parameter must undergo K&R-style default argument promotion.

### 16.8.50 RecordDecl functions

**findFirstNamedDataMember** (*RecordDecl*)  $\Rightarrow$  *FieldDecl*

Returns the first named data member of the provided record if any, otherwise returns *None*.

**getField** (*RecordDecl* *Integral*)  $\Rightarrow$  *FieldDecl*

Returns the specified field of the provided record.

**hasFlexibleArrayMember** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record contains a flexible array member.

**hasVolatileMember** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record has a *volatile* member.

**isAnonymousStructOrUnion** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record is an anonymous *struct* or *union*.

**isInjectedClassName** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record represents an injected class name.

**isLambda** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record is a class describing a lambda function object.

**isOrContainsUnion** (*RecordDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided record is a *union* or (recursively) contains a *union*.

**numFields** (*RecordDecl*)  $\Rightarrow$  *Integral*

Returns the number of fields in the provided record.

### 16.8.51 ReturnStmt functions

**getNRVOCandidate** (*ReturnStmt*)  $\Rightarrow$  *VarDecl*

Returns the variable that might be used for named return value optimization.

**getRetVal** (*ReturnStmt*)  $\Rightarrow$  *Expr*

Returns the return value expression of the provided 'return' statement.

### 16.8.52 StaticAssertDecl functions

**getAssertExpr** (*StaticAssertDecl*)  $\Rightarrow$  *Expr*

Returns the expression used in the provided static assertion.

**getMessage** (*StaticAssertDecl*)  $\Rightarrow$  *StringLiteral*

If the provided static assertion contained a message argument, that argument is returned, otherwise *None* is returned.

**isFailed** (*StaticAssertDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided static assertion failed.

### 16.8.53 Stmt functions

**dump** (*Stmt*)  $\Rightarrow$  *String*

Returns a string containing the AST tree of the provided statement.

**getBeginLoc** (*Stmt*)  $\Rightarrow$  *Location*

Returns the Location corresponding with the beginning of the provided statement.

**getEndLoc** (*Stmt*)  $\Rightarrow$  *Location*

Returns the Location corresponding with the end of the provided statement.

**getStmtClassName** (*Stmt*)  $\Rightarrow$  *String*

Returns a string representing the kind of the provided statement.

### 16.8.54 String functions

**endsWith** (*String String*)  $\Rightarrow$  *Boolean*

Returns true if the first argument ends with the second argument.

**length** (*String*)  $\Rightarrow$  *Integral*

Returns the length of the string provided as the first argument.

**lowerCase** (*String*)  $\Rightarrow$  *String*

Returns a copy of the provided string with all characters converted to lowercase.

**startsWith** (*String String*)  $\Rightarrow$  *Boolean*

Returns true if the first argument starts with the second argument.

**strftime** (*String*)  $\Rightarrow$  *String*

Returns a string representing the current date and time according to the provided format string as per the Standard C++ 11 `strftime` function.

**upperCase** (*String*)  $\Rightarrow$  *String*

Returns a copy of the provided string with all characters converted to uppercase.

### 16.8.55 SwitchCase functions

**getNextSwitchCase** (*SwitchCase*)  $\Rightarrow$  *SwitchCase*

Returns the next switch case which is typically the one written prior to the provided one in the source code.

**getSubStmt** (*SwitchCase*)  $\Rightarrow$  *Stmt*

Returns the substatement of the provided switch case.

### 16.8.56 SwitchStmt functions

**getBody** (*SwitchStmt*)  $\Rightarrow$  *Stmt*

Returns the statement comprising the body of the provided `switch` statement.

**getCond** (*SwitchStmt*)  $\Rightarrow$  *Expr*

Returns the condition expression of the provided `switch` statement.

**getConditionVariable** (*SwitchStmt*)  $\Rightarrow$  *VarDecl*

Returns the condition variable declared in the `switch` statement if any, otherwise returns `None`.

**getConditionVariableDeclStmt** (*SwitchStmt*)  $\Rightarrow$  *DeclStmt*

If the provided `while` statement declares a condition variable, returns the declaration statement associated with the condition variable, otherwise returns `None`.

**getFirstSwitchCase** (*SwitchStmt*)  $\Rightarrow$  *SwitchCase*

Returns the switch case at the beginning of the *list* of switch cases for the provided `switch` statement which is typically the last switch case written in the source code for the `switch`.

**getInit** (*SwitchStmt*)  $\Rightarrow$  *Stmt*

If the provided `switch` statement contains an initialization clause, returns the statement that serves as that clause, otherwise returns `None`.

**isAllEnumCasesCovered** (*SwitchStmt*)  $\Rightarrow$  *Boolean*

Returns true if the switch condition has enumeration type and every value associated with the corresponding declared enumerators are covered by cases in the switch.

**16.8.57 TagDecl functions****getCanonicalDecl** (*TagDecl*) ⇒ *TagDecl*

Returns the canonical declaration of the provided tag declaration.

**getDefinition** (*TagDecl*) ⇒ *TagDecl*Returns the tag declaration that defines the provided tag if any, otherwise returns *None*.**getInnerLocStart** (*TagDecl*) ⇒ *Location*

Returns the location of the provided tag declaration ignoring enclosing template declarations.

**getOuterLocStart** (*TagDecl*) ⇒ *Location*

Returns the location of the provided tag declaration considering enclosing template declarations.

**getTypedefNameForAnonDecl** (*TagDecl*) ⇒ *TypedefNameDecl*Returns the typedef declared with the provided anonymous tag declaration if any, otherwise returns *None*.**isClass** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares a class.

**isCompleteDefinition** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag has a complete definition.

**isCompleteDefinitionRequired** (*TagDecl*) ⇒ *Boolean*

Returns true if a complete definition of the provided tag is needed in the current translation unit.

**isDependentType** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares a type that depends on template parameters.

**isEmbeddedInDeclarator** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag is embedded in a declarator.

**isEnum** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares an enumeration.

**isFreeStanding** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag is freestanding.

**isInterface** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares an interface.

**isStruct** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares a struct.

**isThisDeclarationADefinition** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declaration is a definition.

**isUnion** (*TagDecl*) ⇒ *Boolean*

Returns true if the provided tag declares a union.

**16.8.58 TypeDecl functions****getTypeForDecl** (*TypeDecl*) ⇒ *ASTType*

Returns the type of the provided type declaration.

**16.8.59 TypedefNameDecl functions****getAnonDeclWithTypedefName** (*TypedefNameDecl*) ⇒ *TagDecl*Returns the anonymous tag declaration associated with the provided typedef name declaration if any, otherwise returns *None*.**getUnderlyingType** (*TypedefNameDecl*) ⇒ *ASTType*

Returns the underlying type of the provided typedef name declaration.

**16.8.60 UnaryOperator functions****getOpcodeStr** (*UnaryOperator*) ⇒ *String*

Returns the string representation of the provided unary operator.

**isArithmeticOp** (*UnaryOperator*) ⇒ *Boolean*

Returns true if the provided unary operator is an arithmetic operator.

**isDecrementOp** (*UnaryOperator*) ⇒ *Boolean*

Returns true if the provided unary operator is the prefix or postfix decrement operator.

**isIncrementOp** (*UnaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided unary operator is the prefix or postfix increment operator.

**isPostfix** (*UnaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided unary operator is the postfix increment operator or the postfix decrement operator.

**isPrefix** (*UnaryOperator*)  $\Rightarrow$  *Boolean*

Returns true if the provided unary operator is the prefix increment operator or the prefix decrement operator.

### 16.8.61 UsingDecl functions

**hasTypename** (*UsingDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided using declaration uses the `typename` keyword.

**isAccessDeclaration** (*UsingDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided using declaration is a C++ 03 access declaration.

### 16.8.62 UsingDirectiveDecl functions

**getNominatedNamespace** (*UsingDirectiveDecl*)  $\Rightarrow$  *NamespaceDecl*

Returns the namespace nominated by the provided using directive declaration.

### 16.8.63 ValueDecl functions

**getType** (*ValueDecl*)  $\Rightarrow$  *ASTType*

Returns the type of the provided value declaration.

### 16.8.64 ValueStmt functions

**getExprStmt** (*ValueStmt*)  $\Rightarrow$  *Expr*

Returns the expression statement of the provided value statement.

### 16.8.65 VarDecl functions

**getAnyInitializer** (*VarDecl*)  $\Rightarrow$  *Expr*

Returns the initialized attached to any declaration of the provided variable.

**getDescribedVarTemplate** (*VarDecl*)  $\Rightarrow$  *VarTemplateDecl*

Returns the variable template that is described by the provided variable or `None`.

**getInit** (*VarDecl*)  $\Rightarrow$  *Expr*

Returns the initializer attached to the provided variable declaration if any, otherwise returns `None`.

**getInstantiatedFromStaticDataMember** (*VarDecl*)  $\Rightarrow$  *VarDecl*

If the provided variable is a static data member of a class template specialization, returns the templated static data member from which it was instantiated, otherwise returns `None`.

**getTemplateInstantiationPattern** (*VarDecl*)  $\Rightarrow$  *VarDecl*

If the provided variable is an instantiation, returns the variable from which it could have been instantiated, otherwise returns `None`.

**hasExternalStorage** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable has extern storage.

**hasGlobalStorage** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variables does not have local storage.

**hasICEInitializer** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is initialized with an integer constant expression.

**hasInit** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true the provided variable declaration has an initializer.

**hasLocalStorage** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a non-static local variable with function scope.

**isCXForRangeDecl** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a for-range declaration in a for-range statement.

**isConstexpr** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is `constexpr`.

**isDirectInit** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is initialized with a direct-initializer (list or call style initialization).

**isExceptionVariable** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is the exception variable of a `catch` statement.

**isExternC** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable has external C linkage.

**isFileVarDecl** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable has file scope.

**isFunctionOrMethodVarDecl** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a local, non-parameter variable not declared inside of block.

**isInExternCContext** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is subject to a C++ extern "C" linkage specifier.

**isInExternCXXContext** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is subject to a C++ extern "C++ " linkage specifier.

**isInitCapture** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is the implicit variable for a lambda init-capture.

**isInline** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a C++ 20 `inline` variable.

**isInlineSpecified** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the `inline` keyword was specified for the provided variable.

**isLocalVarDecl** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a local, non-parameter variable.

**isLocalVarDeclOrParm** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a local or function-parameter variable.

**isNRVOVariable** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a local variable that may be used for named return value optimization.

**isParameterPack** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a function parameter pack or init-capture pack.

**isStaticDataMember** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a static data member of a class.

**isStaticLocal** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable is a static local variable with function scope.

**isUsableInConstantExpressions** (*VarDecl*)  $\Rightarrow$  *Boolean*

Returns true if the provided variable may be use in constant expressions.

### 16.8.66 WhileStmt functions

**getBody** (*WhileStmt*)  $\Rightarrow$  *Stmt*

Returns the statement comprising the body of the provided `while` statement.

**getCond** (*WhileStmt*)  $\Rightarrow$  *Expr*

Returns the condition expression of the provided `while` statement.

**getConditionVariable** (*WhileStmt*)  $\Rightarrow$  *VarDecl*

Returns the condition variable declared in the `while` statement if any, otherwise returns `None`.

**getConditionVariableDeclStmt** (*WhileStmt*)  $\Rightarrow$  *DeclStmt*

If the provided `while` statement declares a condition variable, returns the declaration statement associated with the condition variable, otherwise returns `None`.



## 17 Other Features

### 17.1 Format Checking

The `printf`-like and `scanf`-like functions are fertile ground for programming errors as they are not type safe due to their variadic nature, and incorrect use often involves undefined behavior that is not diagnosed by compilers. PC-lint Plus performs comprehensive analysis of the use of these functions diagnosing format incompatibilities, inconsistent and redundant specifier combinations, missing and unused arguments, mis-use of positional specifiers, use of non-standard conversion specifiers, unbounded conversions, and other anomalies.

There are several categories of checking performed and over two dozen messages dedicated to analysis of format string functions.

#### 17.1.1 Dangerous Use

The messages in this section focus on particularly egregious errors that always have the potential to result in undefined behavior. A relatively common error is to provide fewer data arguments than required by the format string. This often happens for particularly large format strings or when the format string is changed. Another possibility is a missing comma between string literal arguments such as in:

```
printf("\%10s %s", "Name" "Value");
```

which will be diagnosed with:

```
warning 558: too few data arguments for format string (1 missing)
printf("\%10s %s", "Name" "Value");
      ~^
```

showing the location of the first conversion specifier without a value and the total number of missing data arguments. Such a call results in undefined behavior as `printf` processes data on the stack looking for the next argument.

The `scanf`-like functions have additional potential concerns. Using the `%s` or `%[` conversion specifier without a maximum field width will result in undefined behavior if the stored string exceeds the provided buffer, e.g.:

```
char buf[10];
scanf("%s", buf);
```

which will result in:

```
warning 498: unbounded scanf conversion specifier 's' may result in buffer overflow
scanf("%s", buf);
      ~^
```

This message will not be issued if the non-standard `'m'` prefix is used (e.g. `%ms`), which specifies that `scanf` should dynamically allocate a buffer large enough to hold the result.

A missing closing bracket for the `%[` conversion specifier is yet another instance of undefined behavior and is diagnosed with message [2406](#). For example:

```
char buf[100];
scanf("%99[~]", buf);
```

Here the programmer intended to store a series of consecutive `~` characters into `buf` but a special exception for the `scanf` function causes the closing bracket to be considered part of the pattern in this case, not the closing bracket to the `%[` conversion specifier. PC-lint Plus will issue:

```
warning 2406: no closing ']' for '%[' in scanf format string
scanf("%99[~]", buf);
      ~~~
```

Another source of undefined behavior stemming from `scanf` is when a field width of zero is specified:

```
scanf("%0s", buffer);
```

which will elicit:

```
warning 2407: zero field width in scanf format string is unused
scanf("%0s", buffer);
      ^
```

Less common is a format string that is not null terminated, this can happen when a sized array of `char` is initialized in a way that prevents the terminating NUL character from being appended, e.g.:

```
const char fmt[2] = "%s";
```

which when used as a format string will result in warning 496 (the declaration alone is enough to prompt info 784).

### 17.1.2 Argument Inconsistencies

This group of messages will diagnose arguments to format functions that do not match the corresponding conversion specifier in the format string. Warning 559 is issued for significant discrepancies:

```
printf("\%s", 12);
```

will elicit:

```
warning 559: format '%s' specifies type 'char *' which is inconsistent with
argument no. 2 of type 'int'
printf("\%s", 123);
      ~ ~    ^ ~ ~
```

Messages 705 and 706 are used to diagnose "nominal" inconsistencies between the expected and actual type or the type pointed to. A "nominal" difference means that the size and basic type of the argument was correct but the type was not exactly the type prescribed by the Standard, e.g. a difference in sign. `printf`-like functions can accept a star (\*) in place of the field width and/or precision in which case the value is taken from the next `int` argument. If this argument is missing warning 2402 is issued. Warning 2403 is issued if the type of this argument is incorrect.

### 17.1.3 Positional Arguments

The POSIX positional argument syntax allows arguments to be referred to by number using the syntax `n$` where `n` refers to a data argument. For example the following two calls to `printf` are equivalent:

```
printf("%d %d", 1, 2);      // ISO syntax
printf("%1$d %2$d", 1, 2); // POSIX positional notation
```

Positional arguments allow format strings to reference arguments in an order that is different from how they are supplied to the format function as well as using the same argument multiple times in the format string. There are several caveats when using positional arguments. For starters, the position starts at 1, not 0; an attempt to use 0 as a position will elicit message 493. Positional arguments cannot be mixed with non-positional arguments in the same format string, violations of this rule are diagnosed by message 2401. Referencing a non-existent positional argument will be diagnosed by messages 494 (data argument positions) and 2404 (field width and field precision positions).

### 17.1.4 Non-ISO features

Features that are not specified by the ISO C Standard may not be portable to other platforms and their use can be diagnosed by PC-lint Plus. These features include positional arguments described above (message 855), non-ISO format specifiers such as `%m` for `printf`-like functions and `%C`, and `%D` for `printf`-like and `scanf`-like functions (message 816), and non-standard length modifiers / conversion specifier combinations (message 499). Since the behavior of these features are not specified by the Standard, their use on platforms that do not support them may result in unintended or undefined behavior.

### 17.1.5 Incorrect Format Specifiers

There are several reasons that a conversion specifier may be invalid and PC-lint Plus will diagnose these.

1. An incomplete format specifier (e.g. `%h`) will be diagnosed by warning [492](#),
2. an unknown conversion specifier (e.g. `%b`) by warning [557](#),
3. inconsistent or redundant format specifiers (e.g. `%+u`) are diagnosed by warning [566](#), and
4. illegal use of a field width or precision with a conversion specifier will result in warning [2405](#).

Each of these messages represent a programming error or the use of extensions that PC-lint Plus is not aware of. For example:

```
printf("\%.10c", 'a');
```

will result in:

```
warning 2405: precision used with 'c' conversion specifier is undefined
printf("\%.10c", 'a');
~~~~~
```

A precision is not allowed with the `%c` conversion specifier and providing one results in undefined behavior.

### 17.1.6 Suspicious Format Specifiers

There are several suspicious constructs that by themselves do not represent errors but are sufficiently unusual to warrant review. This includes

1. an empty format string (message [497](#)),
2. a format string that contains an embedded NUL character (message [495](#)),
3. the use of a non-literal format string (messages [592](#) and [905](#)),
4. unused data arguments (message [719](#)).

### 17.1.7 Elective Notes and Customization

Message [983](#) will point out uses of dash(-) within a `scanf` scan-list (e.g. `%[A-Z]`). As the behavior of the dash in this position is implementation defined, some implementations interpret this as a range, others do not.

`printf` and `scanf` conversion specifiers can also be deprecated using the `-deprecate` option, which will cause message [586](#) to be emitted when they are seen. E.g. `-deprecate(printf_code, n)` will cause a warning to be issued whenever the `%n` conversion specifier is used in a `printf`-like function. Similarly, use `scanf_code` to deprecate `scanf` conversion specifiers. See `-deprecate` for additional information.

The `-printf` and `-scanf` options allow a user to specify functions that resemble a member of the `printf` or `scanf` family. PC-lint Plus has built-in support for the following formatting functions, including those from Annex K in the C11 Standard:

printf-like functions	scanf-like functions	Annex K printf-like functions	Annex K scanf-like functions
fprintf	fscanf	fprintf_s	fscanf_s
fwprintf	fwscanf	fwprintf_s	fwscanf_s
printf	scanf	printf_s	scanf_s
snprintf	sscanf	snprintf_s	sscanf_s
sprintf	swscanf	snwprintf_s	swscanf_s
swprintf	vfscanf	sprintf_s	vfscanf_s
vwprintf	vscanf	swprintf_s	vwscanf_s
vprintf	vsscanf	vfprintf_s	vscanf_s
vsnprintf	wscanf	vfwprintf_s	vsscanf_s
vsprintf		vprintf_s	vwscanf_s
wprintf		vsnprintf_s	vwscanf_s
		vsnwprintf_s	wscanf_s
		vsprintf_s	
		vswprintf_s	
		vwprintf_s	
		wprintf_s	

## 17.2 Precision, Viable Bit Patterns, and Representable Values

Several messages (including [650](#), [587](#), and [734](#)) deal with the notion of “precision” or otherwise involve static determination of whether or not a value is representable in a particular context. In PC-lint Plus precision has been expanded from covering only the conceptual width of a value (e.g. a bitfield or right-shifted variable) to encompass the potential bit patterns that can result from the use of bitwise operators, addition, subtraction, or values of `enum` type. Many such messages utilize supplemental messages to convey bit pattern information when relevant. For example, for some `unsigned int u`:

```
if ( (u & 0x10) == 0x11 ) { }
```

will result in:

```
warning 587: predicate '==' can be pre-determined
        and always evaluates to false
```

```
if( (u & 0x10) == 0x11 ) { }
```

~~~~~ ^ ~~~~~

supplemental 891: incompatible bit patterns:

[illegible]

```
U32  00000000000000000000000000000000?0000
```

```
if ( (u & 0x10) == 0x11 ) { }
```

→

[illegible]

In a more complex example, the effects of implicit conversions may be visible, for example:

```

1 void f(char c1, char c2) {
2     if ( (c1 & 13) + (c2 & 8) == 6 ) { }
3 }

```

will result in:

```
warning 587: predicate '==' can be pre-determined and always evaluates to false
```

```
if ( (c1 & 13) + (c2 & 8) == 6 ) { }
```

~~~~~ ^ ~

```

supplemental 891: incompatible bit patterns:
  S32_00000000000000000000000000000000110 vs
  S32_.....??0?
  if ( (c1 & 13) + (c2 & 8) == 6 ) { }
      ^

```

The first bit pattern represents the constant 6. The second bit pattern is masked with periods beyond its meaningful precision because it is signed in order to reduce confusion regarding the potential sign extension of an inexact value. This message is indicating that the resultant sum cannot ever have the bit in the twos place set and therefore cannot represent the value to which it is being compared.

Supplemental messages displaying bit patterns will only appear when they will provide useful information beyond that conveyed by the precision specified in the original message. For example:

```

1 void f(unsigned char uc) {
2     if (uc == -1) { }
3 }

```

will result in:

```

warning 650: constant '-1' out of range for operator '=='
      if (uc == -1) { }
          ^

```

with no accompanying supplemental message.

The following message numbers currently utilize the unified precision and viable bit pattern architecture:

| #    | Context                                                               | Category                 |
|------|-----------------------------------------------------------------------|--------------------------|
| 572  | >> or >>= by a constant                                               | loss of precision        |
| 587  | ==, !=, <, <=, >, or >= with one constant operand                     | pre-determined predicate |
| 650  | ==, !=, <, <=, >, or >= with one constant operand, switch case        | pre-determined predicate |
| 685  | <, <=, >, or >= with one constant operand                             | pre-determined predicate |
| 734  | assignment                                                            | loss of precision        |
| 2415 | ==, !=, <, <=, >, or >= in a loop condition with no constant operands | pre-determined predicate |

### 17.3 Static Initialization

Traditional lint Compilers do not flag uninitialized static (or global) variables because the C/C++ language defines them to be 0 if no explicit initialization is given. But uninitialized statics, because they can cover such a large scope, can be easily overlooked and can be a serious source of error. Additionally, some embedded compilers do not perform this standard mandated implicit initialization. PC-lint Plus will flag static variables (see messages [727](#), [728](#) and [729](#)) that have no initializer and that are assigned no value. For example, consider:

```

int n;
int m = 0;

```

There is no real difference between the declarations as far as C/C++ is concerned but PC-lint Plus regards `m` as being explicitly initialized and `n` not explicitly initialized. If `n` is accessed by nowhere assigned a value, a complaint will be emitted.

### 17.4 Indentation Checking

Indentation checking can be used to locate the origins of missing left and right braces. It can also locate potential problems in a syntactically correct program. For example, consider the code fragment:

```

if( ... )
    if( ... )
        statement
    else statement

```

Apparently the programmer thought that the **else** associates with the first **if** whereas a compiler will, without complaint, associate the **else** with the second **if**. PC-lint Plus will signal that the **else** is negatively indented with respect to the second **if**.

There are three forms of messages; Informational 725 is issued in the case where there is no indentation (no positive indentation) when indentation is expected, Warning 525 is issued when a construct is indented less than (negatively indented from) a controlling clause, and 539 is issued when a statement that is not controlled by a controlling clause is nonetheless indented from it.

Of importance in indentation checking is the weight given to leading tabs in the input file. Leading tabs are by default regarded as 8 blanks but this can be overridden by the **-t#** option. For example **-t4** signifies that a tab is worth 4 blanks (see the **-t#** option in Section 4.3.3 Message Presentation).

Recognizing indentation aberrations comes dangerously close to advocating a particular indentation scheme; this we wish to avoid. For example, there are at least three main strategies for indentation illustrated by the following templates:

```

if( e ) {
    statements
}

if( e )
{
    statements
}

if( e )
{
    statements
}

```

Whereas the indentation methods appear to differ radically, the only real difference is in the way braces are handled. Statements are always indented positively from the controlling clause. For this reason PC-lint Plus makes what is called a *strong* check on statements requiring that they be indented (or else a 725 is issued) and only a *weak* check on braces requiring merely that they not be negatively indented (or else a 525 is issued).

**case**, and **default** undergo a weak check. This means, for example, that

```

switch()
{
case 'a' :
    break;
default:
    break;
}

```

raises only the informational message (725) on the second **break** but no message appears with the **case** and **default** labels.

The **while** clause of a **do ... while(e);** compound undergoes a weak check with respect to the **do**, and an **else** clause undergoes a weak check with respect to its corresponding **if**.

An **else if()** construct on the same line establishes an indentation level equal to the location of the **else** not the **if**. This permits use of the form:

```

if()
    statement}

```

```

else if()
    statement
else if()
    statement
...
else
    statement

```

Only statement beginnings are checked. Thus a comment can appear anywhere on a line and it will not be flagged. Also a long string (if it does not actually begin a statement) may appear anywhere on the line.

A label may appear anywhere unless the `+fil` flag is given (Section 4.11 Flag Options) in which case it undergoes a weak check.

Message 539 is issued if a statement that is not controlled by a loop is indented from it. Thus:

```

while ( n > 0 );
    n = f(n);

```

draws this complaint, as well it should. It appears to the casual reader that, because of the indentation, the assignment is under the control of the `while` clause whereas a closer inspection reveals that it is not.

## 17.5 Size of Scalars

Since the user of PC-lint Plus has the ability to set the sizes of various data objects (See the `-s..` options in Section 4.5.1 Scalar Data Size), the reader may wonder what the effect would be of using various sizes.

Several of the loss of precision messages (712, 734, 735 and 736) depend on a knowledge of scalar sizes. The legitimacy of bit field sizes depends on the size of an `int`. Warnings of format irregularities are based in part on the sizes of the items passed as arguments.

One of the more important effects of type sizes is the determination of the type of an expression. The types of integral constants depend upon the size of `int` and `long` in ways that may not be obvious. For example, even where `int` are represented in 16 bits the quantity:

```
35000
```

is `long` and hence occupies 4 (8-bit) bytes whereas if `int` is 32 bits the quantity is a four byte `int`. If you want it to be `unsigned` use the `u` suffix as in `35000u` or use a cast.

Here are the rules: the type of a decimal constant is the first type in the list (`int`, `long`, `long long`) that can represent the value. The maximum values for these types are taken to be  $2^{\text{sizeof}(\text{type}) * \text{bits-per-byte} - 1} - 1$ . The quantities `sizeof(int)`, `sizeof(long)`, and `sizeof(long long)` are based on the `-si#`, `-sl#`, and `-sll#` options respectively. The type of a hex or octal constant, however, is the first type on the list (`int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`).

For any constant (decimal, hex or octal) with a `u` or `U` suffix, one selects from the list (`unsigned int`, `unsigned long`, `unsigned long long`). If an `l` or `L` suffix, the list is (`long`, `long long`) for decimal constants and (`long`, `unsigned long`, `long long`, `unsigned long long`) for hex and octal constants. If both suffixes are used (e.g. `UL`), the list is (`unsigned long`, `unsigned long long`) for any constant. If the suffix is `ll` or `LL`, the type is `unsigned long long` for decimal constants and either `long long` or `unsigned long long` for hex and octal constants. Finally, constants containing both the `u/U` and `ll/LL` suffixes are always of type `unsigned long long`, regardless of base.

The size of scalars enters into the typing of intermediate expressions in a computation. Following ANSI/ISO standards, PC-lint Plus uses the so-called *value-preserving* rule for promoting types. Types are promoted

when a binary operator is presented with two unlike types and when unprototyped function definitions specify subinteger parameters. For example, if an `int` is added to an `unsigned short`, then the latter is converted to `int` provided that an `int` can hold all values of an `unsigned short`; otherwise, they are both converted to `unsigned int`. Thus the signedness of an expression can depend on the size of the basic data objects.

## 17.6 Stack Usage Report

```
+stack(sub-option,...)
-stack(sub-option,...)
```

The `+stack` version of this option can be used to trigger a stack usage report. The `-stack` version is used only to establish a set of options to be employed should a `+stack` option be given. To prevent surprises if a `-stack` option is given without arguments it is taken as equivalent to a `+stack` option.

The sub-options are:

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&amp;file=filename</code> | This option designates the file to which the report will be written. This option must be present to obtain a report.                                                                                                                                                                                                                                                                                                                                                                   |
| <code>&amp;overhead(n)</code>   | establishes a call overhead of $n$ bytes. The call overhead is the amount of stack consumed by a parameterless function that allocates no <code>auto</code> storage.<br><br>Thus if function <code>A()</code> , whose auto requirements are 10, calls function <code>B()</code> , whose auto requirements are also 10, and which calls no function, then the stack requirements of function <code>A()</code> are $20+n$ where $n$ is the call overhead. By default, the overhead is 8. |
| <code>&amp;external(n)</code>   | establishes an assumption that each external function (that is not given an explicit stack requirement, see below) requires $n$ bytes of stack. By default this value is 32.                                                                                                                                                                                                                                                                                                           |
| <code>&amp;summary</code>       | This option indicates that the programmer is interested in at least a summary of stack usage (stack used by the worst case function). The summary comes in the form of Elective Note <a href="#">974</a> and is equivalent to issuing the option <code>+e974</code> . This option is not particularly useful since a summary report will automatically be given if a <code>+stack</code> option is given. It is provided for completeness.                                             |
| <code>name(n)</code>            | where <i>name</i> is the name of a function, explicitly designates the named function as requiring $n$ bytes of total stack. This is typically used to provide stack usage values for functions whose stack usage could not be computed either because the function is involved in recursion or in calls through a function pointer. <i>name</i> may be a qualified name.                                                                                                              |

Example:

```
+stack( &file=s.txt, alpha(12), A::get(30) )
```

requests a stack report to be written to file `s.txt` and further, that function `alpha()` requires 12 bytes of stack and function `A::get()` requires 30.

At global wrap-up, a record is written to the file for each defined function. The records appear alphabetized by function name.

Each record will contain the name of a function followed by the amount of auto storage required by its local auto variables. Note that auto variables that appear in different and non-telescoping blocks may share storage so the amount reported is not simply the sum of the storage requirements of all auto variables.

Each function is placed into one of seven categories as follows:

1. *recursive loop* – a function is *recursive loop* if it is recursive and we can provide a call to a function such that that call is in a recursive loop that terminates with the original function. Thus the function is not



merely recursive but demonstrably recursive. The record contains the name of a function called and it is guaranteed that the called function will also be reported as *recursive loop*.

It is assumed that any recursive function requires an unbounded amount of stack. If that assumption is incorrect and you can deduce an upper bound of stack usage, then you can employ the `+stack` option to indicate this upper bound. In a series of such moves you can convert a set of functions containing recursion to a set of functions with a known bound on the stack requirements of each function.

2. *recursive* – a function is designated as *recursive* if it is recursive but we do not provide a specific circular sequence of calls to demonstrate the fact. Thus the function is recursive but unlike *recursive loop* functions it is not demonstrably recursive. The record contains the name of a function called. This function will either be *recursive loop*, *recursive* or *calls recursive* (see next category). If you follow the chain of calls it is guaranteed that you will ultimately arrive at a function that is labeled *recursive loop*.

The distinction between the *recursive* and *recursive loop* categories is illustrated by the example:

```
void x(), y(), z();
void x() { y(); z(); }
void y() { x(); }
void z() { x(); }
```

The stack report will show that `x` calls `y`, and that `y` calls `x`. As this demonstrates the recursive nature of both functions, they will be placed in the *recursive loop* category. The report will also show `z` calling `x`, but the displayed callee for the `x` entry is `y`. As the report does not demonstrate the call chain leading to a recursive call to `z`, it will be classified as *recursive*.

3. *calls recursive* – a function may itself be non-recursive but may call a function (directly or indirectly) that is recursive. The stack requirements of functions in this category are considered to be unbounded. The record will contain the name of a function that it calls. This function will either be '*recursive loop*', '*recursive*' or '*calls recursive*'. If you follow the chain of calls it is guaranteed that you will ultimately arrive at a function that is labeled '*recursive loop*'.
4. *non-deterministic* – a function is said to be *non-deterministic* if it calls through a function pointer. The presumption is that we cannot determine by static means the set of functions so called. No function is labeled *non-deterministic* unless it is first determined that it is not in the *recursive* categories. That is, it could not be determined following only deterministic calls that it could reach a *recursive* function.

If you can determine an upper bound for the stack requirements of a non-deterministic function then, like a recursive function, you may employ the `+stack` option to specify this bound and in a sequence of such options determine an upper bound on the amount of stack required by the application.

5. *calls a non-deterministic function* – a function is placed into this category if it calls directly or indirectly a *non-deterministic function*. It is guaranteed that we could not find a recursive loop involving this function or even a deterministic path to a recursive function. The record will be accompanied by the name of a function called. It is guaranteed that if you follow the chain of calls you will reach a non-deterministic function.
6. *finite* – a function is *finite* if all call chains emanating from the function are bounded and deterministic. The record will contain a total stack requirement. This will be a worst case stack usage. The record will bear the name of a function called (or 'no function' if it does not call a function). If you follow this chain you will pass through a (possibly zero length) sequence of finite functions before arriving at a function that

- (a) is labeled as '*finite*' but calls no other function or
- (b) is labeled as '*external*' or
- (c) is labeled as '*explicit*' (see next category).

You should be able to confirm the stack requirements by adding up the contribution from each function in the chain plus a fixed call overhead for each call. The amount of call overhead can be controlled by

the **stack** option.

For *'external'* functions there is an assumed default stack requirement. You may employ the **+stack** option to specify the stack requirement for a specific function or to alter the default requirement for external functions.

7. *explicit* – a function is labeled as *explicit* if there was an option provided to the **-stack** option as to the stack requirements for a specific function.

## Stack Report Formatting Options

The information provided by this option can be formatted by the user using the **-format\_stack** option. This allows the information to be formatted to a form that would allow it to be used as input to a database or to a spreadsheet. This format can contain the escape codes:

- %f** for the function name
- %a** for the local auto storage
- %t** for type (i.e. one of the seven categories above)
- %n** for the total stack requirement (or -1 if unbounded by recursion)
- %c** for the callee and
- %e** for an *'external'* tag on the callee

See **-format\_stack** for more details. See also Message [974](#).

## 17.7 Migrating to 64 bits

Applications written for the traditional 32-bit model where **int**, **long** and pointers are each represented in 32 bits, may have difficulty when ported to one of the 64-bit models. Problems that you may encounter and that PC-lint Plus will catch are described and implemented as options in file **au-64.lnt**. This file and other **.lnt** files mentioned below are distributed with the product and/or are downloadable from our web site.

The file **au-64.lnt** is not intended to be used directly by the programmer. There are a number of wrappers reflecting the different flavors of 64-bit computing. These wrappers specify sizes and other options unique to specific models and then invoke **au-64.lnt**. The models and the **au** file that you should be using are described below.

| Data Type | LP64 Model | LLP64 Model | ILP64 Model |
|-----------|------------|-------------|-------------|
| long long | 64 bits    | 64 bits     | 64 bits     |
| pointers  | 64 bits    | 64 bits     | 64 bits     |
| long      | 64 bits    | 32 bits     | 64 bits     |
| int       | 32 bits    | 32 bits     | 64 bits     |

The above table shows the differences between each of the 3 64-bit models. Each model has a corresponding **au** file: **au-lp64.lnt** for LP64, **au-llp64.lnt** for LLP64, and **au-ilp64.lnt** for ILP64.

## 17.8 Deprecation of Entities

You may indicate that a particular *name* is not to be employed in your programs by using this option:

```
-deprecate(category, name [,commentary])
```

*category* is one of: **function**, **keyword**, **macro**, **option**, **variable**, **type**, **basetype**, **ppw**, **printf\_code** or **scanf\_code**.

The *commentary* in the third argument will be appended to the message. For example,

```
-deprecate( variable, errno, Violates Policy XX-123 )
```

When the use of `errno` as a variable is detected (but not its definition or declaration) the following Warning is issued.

```
Warning 586: variable 'errno' is deprecated. Violates Policy XX-123
```

When the category of deprecation is `variable` only the use of external variables are flagged. Local variables may be employed without disparaging comment.

If `errno` were a macro you would need to deprecate `errno` as a macro:

```
-deprecate( macro, errno, Violates Policy XX-123 )
```

If `errno` could be either (the standard allows both forms) then both options should be used.

You may also deprecate functions and keywords. For example:

```
-deprecate( keyword, goto, goto is considered harmful )
-deprecate( function, strcpy, has been known to cause overruns )
```

could be used to flag the use of suspect features.

Quotes (both single and double) and parentheses within the commentary need to be balanced.

### 17.8.1 Deprecation of Options

Options can also be deprecated. Deprecating an option causes future uses of that option to be met with message [586](#) although the option is still processed as usual. When deprecating an option, you must specify the name of the option, including a leading `'-'` or `'+'` but must not provide any arguments for the option. For example:

```
-deprecate( option, -setenv, environment variables should not be set during the
linting process)
```

will deprecate the use of `-setenv`. It is not possible to deprecate the use of **individual** flag options; using the `-deprecate` option with `-f`, `+f`, `--f`, or `++f` will deprecate **all** flag options. Note that some options have forms that begin with `'-'` and another form that begins with `'+'`; deprecating one form does not automatically cause the other form to be deprecated even if both forms have identical meanings.

### 17.8.2 Deprecation of Types

The `-deprecate` option can be used to deprecate types. This can be accomplished using the deprecation categories `'type'` and `'basetype'`.

When the category `'type'` is specified, message [586](#) is issued for any use of the type (outside of typedef declarations and template instantiations) but type alias names (introduced via `typedef` or `using`) are not looked through and use of the underlying type is allowed without complaint if it occurs through such an alias.

The category of `'basetype'` is similar except type aliases are looked through and if at any level the deprecated type is present, [586](#) is issued. If the deprecated type is a `typedef` type, no diagnostic is issued for the declaration of the type (although use of the type is diagnosed).

Using a deprecated `type` as a target of a `typedef` is not diagnosed with [586](#). The logic is that for `'basetype'`, the use of the `typedef` that targets the deprecated type will be diagnosed anyway and that for `'type'` the user is not interested in use of the type through `typedefs`. We do provide a new elective message, [986](#), that will be issued when the target of a type alias is deprecated with the `'type'` category.

```
//lint -deprecate(type, int) warn about uses of int but not through an alias
//lint -deprecate(basetype, DOUBLE)

typedef int INT;          // 986 issued for 'int' appearing in typedef
typedef double DOUBLE;    // okay, declaration of deprecated type
typedef DOUBLE DOUBLE2;   // okay, use of deprecated basetype in alias

int i;// 586 - 'int' is deprecated
INT i2;// okay, use of deprecated type via alias

union u {
    const int * pci;       // 586 - deprecated type 'int' used in declaration
    double * pd;          // okay, 'double' is not deprecated
    DOUBLE d1;            // 586 - deprecated basetype 'DOUBLE' used directly
    DOUBLE2 d2;           // 586 - deprecated basetype 'DOUBLE' used indirectly
};
```

### 17.8.3 Deprecation of Preprocessor Directives

Individual preprocessor directives can be deprecated with the `ppw` category. For example:

```
-deprecate(ppw, pragma)
```

will deprecate the use of the `#pragma` preprocessor directive, note that the `#` is not included in the `-deprecate` option. Message [586](#) will be issued when the deprecated directive is encountered unless it is encountered in a conditionally excluded region (e.g. between `#if 0 ... #endif`) in which case message [886](#) will be issued instead.

An unknown preprocessor directive can be deprecated, PC-lint Plus will issue the deprecation message and then complain about the unknown directive as usual. The `#error` directive can be deprecated in which case the deprecation message will be emitted before the directive is handled. The null directive can be deprecated using an empty second argument to `-deprecate`.

Note that the `-ppw_asgn` option does not establish any deprecation relationship between the named directives. For example, if the option `-ppw_asgn(foo, line)` is used to cause `#foo` to be handled as `#line`, deprecating `line` will not also deprecate `foo` or vice versa.

### 17.8.4 Deprecation of Format Function Conversion Specifiers

The `-deprecate` option can be used to deprecate conversion specifiers for `printf`-like and `scanf`-like functions using the `printf_code` and `scanf_code` categories. For example,

```
-deprecate(printf_code, n)
```

will deprecate the use of `%n` in `printf`-like functions, note that the `%` is not included in the `-deprecate` option. Deprecating a conversion specifier will result in message [586](#) being issued if the conversion specifier is used, regardless of any length modifiers present in the actual use, but will not deprecate other conversion specifiers with the same meaning. For example `-deprecate(printf_code, i)` will warn for `%i` and `%hi` but will not warn for `%d` (which has the same meaning as `%i` in `printf`-like functions).

## 17.9 Parallel Analysis

PC-lint Plus supports the long-requested feature of utilizing multiple cores to achieve faster processing times. This feature is enabled by placing the new `-max_threads=n` option before the first module to process. If this option does not appear before the first module is seen, the behavior is as if `-max_threads=1` was used. Threads are used both during the main processing phase and the global wrap-up phase. In the main phase, a

separate thread is dispatched to handle each module, up to a maximum of  $n$  concurrent threads. When all of the modules have been processed, threads are employed to handle wrap-up processing, again up to a maximum of  $n$  concurrent threads.

When using C++20 Modules with  $n > 1$ , if one or more threads are waiting to process a module import pending the completion of the build of a module interface unit in another thread, then a single extra thread may also be launched to process another module in the meantime. This will never result in more than  $n$  threads concurrently performing work, but both waiting threads and the single extra thread contribute to the overall process memory utilization with the potential peak memory usage being similar to what would otherwise be expected from a fractional value somewhere between  $n$  and  $n+1$  threads.

While results will vary depending on a variety of factors, the best overall times are typically achieved when using a value for  $n$  that equals the number of available cores, or about twice the number of cores for processors that support hyper-threading. Some experimentation may be necessary to find the best value for  $n$  on a particular system. In order to assist in that regard, the option `-max_threads=0` will result in PC-lint Plus picking a value for  $n$  that it thinks is optimal based on querying of the available hardware for systems that support it. When using `-max_threads=0`, elective note 999 will report on the number of threads that has been selected.

There are a few caveats to keep in mind when employing multiple threads:

1. Very little memory is shared between threads, which means that memory usage scales roughly linearly with the number of concurrent threads. For example, if using 1 thread results in memory usage of 500MB, it probably wouldn't be productive to utilize more than 4 threads on a system with 2GB of RAM, regardless of how many cores may be available.
2. Output is buffered by module when using multiple threads. This means that the output for a module will not be emitted until the entire module is processed (this happens before global wrap-up). Additionally, the order in which modules are processed is not guaranteed although output will never be interleaved between modules. For example, when processing modules A and B, the diagnostics for module A may appear (in their entirety) before or after module B when using multiple threads and this order may change between runs. When using a single thread, diagnostics for one module will always appear before the diagnostics for a later-provided module.
3. Interactive features such as the Value Tracking Debugger are supported only when executing with a single thread.

Aside from the above caveats, there is no difference in behavior or functionality when using multiple threads.

## 17.10 Language Limits

The C and C++ Standards define minimum translation limits that must be supported by a conforming compiler. The limits specify quantities such as the minimum number of significant characters in internal and external identifiers, the minimum number of function parameters that an implementation must support, the minimum number of supported concurrently defined macros, and the minimum number of data members supported in structures. The C99 limits are specified in section 5.4.2.1 of the C99 Standard (ISO/IEC 9899:1999) and the C++ limits are specified in Annex B of the C++ Standard (ISO/IEC 14882:2011).

`-lang_limit(C\C++, limit-name, limit-value)` specify minimum language translation limits

The `-lang_limit` option takes three arguments. The first argument is either C or C++ indicating which language the overridden limit applies to. The second argument is the type of limit and must be one of the names in the first column of the table below. The last argument is a value that must be between 0 and 4294967294 (0 indicates the lack of a limit) or the special value of `default`, which reverts back to the corresponding value of the table below essentially removing a previously overridden value.

The table below lists the language limit checks supported by PC-lint Plus. The Limit Name is the name recognized in the second parameter of the `-lang_limit` option. The Limit Description is the text that is used in the 793 message when the limit is exceeded and which can be used with the `-estring` option for suppression purposes. The C89 Limit shows the minimum limits specified by the ANSI C89 standard. The C99 Limit column shows the minimum limits mandated by C99, the C11 Limits are identical. The C++ Limit shows the limits required by the C++ Standard (all versions of C++ share the same limits).

| Limit Name                             | Limit Description                                | C89<br>Limit | C99<br>Limit | C++<br>Limit |
|----------------------------------------|--------------------------------------------------|--------------|--------------|--------------|
| <code>external_identifiers</code>      | external identifiers                             | 511          | 4095         | 65526        |
| <code>internal_identifier_chars</code> | significant characters in an internal identifier | 31           | 63           | 1024         |
| <code>macro_identifier_chars</code>    | significant characters in a macro name           | 31           | 63           | 1024         |
| <code>external_identifier_chars</code> | significant characters in an external identifier | 6            | 31           | 1024         |
| <code>function_parameters</code>       | function parameters                              | 31           | 127          | 256          |
| <code>function_arguments</code>        | function arguments                               | 31           | 127          | 256          |
| <code>macro_parameters</code>          | macro parameters                                 | 31           | 127          | 256          |
| <code>string_literal_length</code>     | characters in a string literal                   | 509          | 4095         | 65536        |
| <code>case_labels</code>               | case labels in a switch                          | 257          | 1023         | 16384        |
| <code>structure_members</code>         | structure members                                | 127          | 1023         | 16384        |
| <code>enumeration_constants</code>     | enumeration constants                            | 127          | 1023         | 4096         |
| <code>base_classes</code>              | base classes                                     | n/a          | n/a          | 16384        |
| <code>direct_base_classes</code>       | direct base classes                              | n/a          | n/a          | 1024         |
| <code>class_members</code>             | members in a class                               | n/a          | n/a          | 4096         |
| <code>static_members</code>            | static members in a class                        | n/a          | n/a          | 1024         |
| <code>final_functions</code>           | final overriding virtual functions in a class    | n/a          | n/a          | 16384        |
| <code>virtual_base_classes</code>      | virtual base classes                             | n/a          | n/a          | 1024         |
| <code>friend_decls</code>              | friend declarations in a class                   | n/a          | n/a          | 4096         |
| <code>access_decls</code>              | access control declarations in a class           | n/a          | n/a          | 4096         |
| <code>ctor_initializers</code>         | member initializers in a constructor             | n/a          | n/a          | 6144         |
| <code>scope_qualifiers</code>          | scope qualifiers in an identifier                | n/a          | n/a          | 256          |
| <code>template_arguments</code>        | template arguments in a template                 | n/a          | n/a          | 1024         |
| <code>try_handlers</code>              | handlers in a try block                          | n/a          | n/a          | 256          |
| <code>throw_specs</code>               | throw specifications in a function               | n/a          | n/a          | 256          |

By default, the limits shown above are used to determine when a minimum limit has been exceeded and, for C, is dependent on the version of the language used. If your compiler supports different limits, or if you just want to be alerted when a different threshold is reached for a particular limit, you can use the `-lang_limit` to override the defaults shown above.

The table below provides additional details about some of the limits checked by PC-lint Plus.

| Limit Name                        | Notes                                                                                                                                                                                                                                  |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>external_identifiers</code> | External identifiers are functions and variables with external linkage.                                                                                                                                                                |
| <code>internal_identifiers</code> | Internal identifiers are any non-preprocessor (e.g. macro) symbols that are not external identifiers. These include type names, class names, local variables, enumeration constants, etc.                                              |
| <code>case_labels</code>          | This does not include the <code>default</code> label or case labels of nested switches.                                                                                                                                                |
| <code>structure_members</code>    | This includes only non-static data members but includes members inherited from base classes.                                                                                                                                           |
| <code>base_classes</code>         | The number of bases for a given class including indirect and virtual bases.                                                                                                                                                            |
| <code>direct_base_classes</code>  | Virtual and non-virtual direct base classes for a class.                                                                                                                                                                               |
| <code>class_members</code>        | Includes all static and non-static data and function members declared directly in the class (e.g. not inherited members). Note that this also includes implicitly generated functions such as constructors, assignment operators, etc. |
| <code>static_members</code>       | All static data and function members, including inherited members for a class.                                                                                                                                                         |
| <code>virtual_base_classes</code> | Direct and indirect virtual base classes for a class.                                                                                                                                                                                  |
| <code>access_decls</code>         | The number of access control (or using) declarations present in a class. Does not include base classes. This is not a count of the <i>access-specifiers</i> present in a class.                                                        |
| <code>ctor_initializers</code>    | The number of items initialized in a constructor member initializer list. Includes base class initializers.                                                                                                                            |
| <code>scope_qualifiers</code>     | This is the number of nested name specifiers present in a <i>qualified-id</i> , e.g. <code>A::B::C</code> contains two scope qualifiers.                                                                                               |

## 18 Preprocessor

### 18.1 Preprocessor Symbols

PC-lint Plus supports several predefined macros including those defined by ISO C and those supported by various compilers. The special behavior of any of these macros will be removed for the remainder of the module if the macros are explicitly defined or undefined using `#define` or `#undef` and permanently removed if defined or undefined with the `-d/+d/++d` or `-u/--u`.

- `_lint` – The special preprocessor symbol `_lint` is pre-defined with a value representing the version of PC-lint Plus. The primary purpose of this symbol is to enable the programmer to determine whether PC-lint Plus is processing the file.

For example, if you have a section of code that is unacceptable to PC-lint Plus for some reason (such as in-line assembly code), you can use `_lint` to make sure that PC-lint Plus doesn't see it. Thus,

```
#ifndef _lint
...
Unacceptable coding sequence
...
#endif
```

will cause PC-lint Plus to skip over the elided material.

The value of `_lint` is  $1000 * \text{Major Version Number} + 10 * \text{Minor Version number} + \text{the Patch Level}$ .

| <i>Version</i> | <i>Value of _lint</i> |
|----------------|-----------------------|
| 1.0.0          | 1000                  |
| 1.0.1          | 1001                  |
| 1.1.0          | 1010                  |
| 1.2.1          | 1021                  |
| 1.12.3         | 1123                  |
| 2.0.0          | 2000                  |

E.g.

```
#if _lint >= 900
    // use Version 9 feature
#endif
#if _lint != 902
    // not for Version 9.02
#endif
```

- `__cplusplus` – This symbol is defined for each module that is interpreted as being a C++ module and is otherwise undefined. The value that this symbol expands to is dependent on the C++ language mode: the value is 201402L for C++14, 201103L for C++11, and 199711L for C++03. C++ modules are determined by extension and possibly by option. See [Chapter 3 The Command Line](#).
- `__COUNTER__` – This macro expands to an integer that automatically increments every time the macro is expanded within a module. The first result of the first expansion in a module is 0, the second expansion is 1, etc. When used with the `##` operator, this macro provides a mechanism to generate unique identifiers.
- `__BASE_FILE__` – Expands to a string literal that contains the name of the module being processed, as the name was provided to PC-lint Plus.



- `__INCLUDE_LEVEL__` – Expands to a non-negative integer representing the `#include` nesting depth in which the macro appears. A value of 0 indicates a non-header location.
- `__TIMESTAMP__` – Expands to a string literal that contains the last modification date and time of the file in which the macro appears, as returned by the `asctime` function. If the modification time information cannot be determined, expands to the string literal "??? ?? ??:?:?:?? ????".
- The following pre-defined identifiers begin and end with double underscore and are ANSI/ISO compatible.
  - `__TIME__` – The current time
  - `__DATE__` – The current date
  - `__FILE__` – The current file
  - `__LINE__` – The current line number
  - `__STDC__` – Defined to be 1.
  - `__STDC_VERSION__` – This is undefined for C++. It is defined for C by default as '199901L'. If you select an earlier version of C using the `-std` option as in `-std=c90` this will be undefined.
  - `__STDC_HOSTED__` – This is defined whenever `__STDC_VERSION__` is defined. When defined it is defined to be 0.

Compiler-dependent preprocessor symbols may also be established as described in [Section 4.12 Compiler Adaptation](#).

## 18.2 #include Processing

When a `#include "filename"` directive is encountered

1. there is first an attempt to `fopen` the named file. But what is the named file? If the `fdi` flag is OFF the name between quotes is used. If the `fdi` flag is ON, the name of the including file is examined to determine the directory. This directory is prefixed to *filename*. The directory of the including file is found by scanning backward for one of possibly several system-related special characters. If the `fopen` fails, we go to step 2.
2. there is an attempt to prepend (in turn) each of the directories associated with options of the form:
 

```
-idirectory
```

 in the order in which the options were presented. If this fails we go to step 3.
3. On systems supporting environment variables, each directory in the sequence of directives specified by the INCLUDE environment variable is prepended to the file.
4. There is an attempt to `fopen` the file by the name provided, without considering flag `fdi`.

If the include directive is of the form

```
#include <filename>
```

then the processing is the same except that step 1 is bypassed.

### 18.2.1 INCLUDE Environment Variable

The INCLUDE environment variable may specify a search path in which to search for header files (`#include` files). For example:

```
set INCLUDE=b:\include;d:\extra
```

specifies that, should the search for a `#include` file within the current directory fail, a search will be made in the directory `b:\include` and, on failing that, a search will be made in the directory `d:\extra`. This searching is done for modules as well as `#include` files. You may select an environment variable other than INCLUDE. See the `-incvar` option.

Notes:

1. No blank may appear between 'INCLUDE' and '='. Blanks adjacent to semicolons (;) are ignored. All other blanks are significant
2. A terminating semi-colon is ignored.
3. This facility is in addition to the `-i...` option and is provided for compatibility with a number of compilers in the MS-DOS environment.
4. Any directory specified by a `-i` directive takes precedence over the directories specified via the `INCLUDE` environment variable.

## 18.3 ANSI/ISO Preprocessor Facilities

ANSI/ISO preprocessing is assumed throughout. If the K&R preprocessor flag is set (`+fkp`) the use of ANSI/ISO (over K&R) is flagged.

### 18.3.1 #line and #

A C/C++ preprocessor may place `#line` directives within C/C++ source code so that compilers (and other static analyzers such as PC-lint Plus) can know the original file and original line numbers that produced the text actually being read. In this way, these processors can report errors in terms of the original file rather than in terms of the intermediate text.

By default, `#line` directives are processed. To ignore `#line` directives use the option `-fln`. Some systems support `#` as an abbreviation for `#line`; these are treated equivalently by PC-lint Plus.

## 18.4 Non-Standard Preprocessing

Preprocessor commands in this section need to be activated via the `+ppw` option. Also, their semantics may be copied via the `ppw_asgn` option.

### 18.4.1 #import

This preprocessor directive is intended to support the Microsoft preprocessor directive of the same name. For example:

```
#import "c:\compiler\bin\x.lib"
```

will determine the base name (in this case "x") and attempt to include, as a header file, `basename.tlh`. Thus, for linting purposes, this directive is equivalent to:

```
#include "x.tlh"
```

Options that may accompany `#import` are ignored. When the (Microsoft) compiler encounters a `#import` directive it will generate an appropriate `.tlh` file if a current one does not already exist. PC-lint Plus will not generate this file.

When compiling, it is possible to place the generated `.tlh` file in a directory other than the directory of the importing file. If this option is chosen, then when linting, this other directory needs to be identified with a `-i` option or equivalent.

This preprocessor word is not enabled by default. It can be enabled via the `+ppw(import)` option. This option has been placed into the various compiler options files for the Microsoft C/C++ compiler.

### 18.4.2 `#include_next`

`#include_next` is supported for compatibility with the GNU C/C++ compiler. It uses the same arguments as `#include` but starts the header file search in the directory just after the directory (in search order sequence) in which the including file was found. See Section 18.2 [include Processing](#) for a specification of the search order.

For example; suppose you place a file called `stdio.h` in a directory that is searched before the compiler's directory. Thus you could intercept the `#include` of `stdio.h` and effectively augment its contents as follows:

```
stdio.h:
```

```
#include_next <stdio.h>
... augmentation
```

### 18.4.3 `#ident`

This directive takes a single argument, a string constant. This directive will cause some compilers to copy the string into an implementation defined portion of the resulting object file. PC-lint Plus processes but ignores this directive.

### 18.4.4 `#sccs`

This is treated identically to `#ident`.

### 18.4.5 `#warning`

The `#warning` directive is used by some compilers to emit a user-defined warning when the directive is reached during preprocessing. It is similar to `#error` but doesn't terminate processing. If this keyword is enabled, PC-lint Plus will issue warning 490 along with the contents of the line that follows the `#warning` directive, in the same manner as for `#error`. In particular, the text that follows is emitted as written except that multiple space characters are collapsed into a single space and macros are not expanded. The text does not need to be a string constant. If your compiler calls this directive `#warn`, you can use

```
+ppw(warning)
-ppw_asgn(warn, warning)
```

to cause PC-lint Plus to support the alternate spelling.

## 18.5 User-Defined Keywords

PC-lint Plus might stumble over strange preprocessor commands that your compiler happens to support. For example, some Unix system compilers support `#assert`. Since this is something that can NOT be handled by a suitable `#define` of some identifier we have added the `+ppw(command-name)` option ('ppw' is an abbreviation for PreProcessor Word). For example, `+ppw(ident)` will add the preprocessor command alluded to above. PC-lint Plus recognizes and ignores the construct.

## 18.6 Preprocessor `sizeof`

The non-standard use of `sizeof` in a preprocessor conditional is supported by some older and embedded compilers but not directly supported by PC-lint Plus because the information necessary to evaluate a `sizeof` is not available during the preprocessing phase. For portability reasons, such constructs should not be used in new code, favoring static assertions or similar mechanisms instead.

For legacy code, we provide a work-around using the new `-pp_sizeof(Text, Value)` option that can be used to direct PC-lint Plus on how to evaluate a particular `sizeof` expression appearing in a preprocessor conditional. *Text* is the text of the expression appearing inside of the `sizeof` and *Value* is the integral value

to apply to the evaluation of the `sizeof` expression. The new warning 2491 is issued when a preprocessor `sizeof` is encountered with an expression that has not been registered with `-pp_sizeof`. This message can be used to identify expressions that need to be registered as well as the text to use for *Text*. For example:

```
#if sizeof(int) < 4
#error "int is too small"
#endif
```

will elicit:

```
test_ppsizeof.c 1 warning 677: sizeof used within preprocessor statement
#if sizeof(int) < 4
^
test_ppsizeof.c 1 warning 2491: unknown expression 'int' in sizeof will
      evaluate to 0, use -pp_sizeof to change the value used for evaluation
test_ppsizeof.c 2 error 309: #error "int is too small"
#error "int is too small"
^
```

The 677 message warns of the use of `sizeof` in the preprocessor, warning 2491 alerts the programmer that PC-lint Plus doesn't know how to handle `sizeof(int)` in the preprocessor and provides direction for using the `-pp_sizeof` option. Message 309 is issued for the failed `#error` directive resulting from the fact that the unknown `sizeof` expression was evaluated to be 0.

To employ the work-around, determine what the value of `sizeof(int)` is on the target platform and register the expression with `-pp_sizeof`. E.g. if `int` is 4 bytes, use `-pp_sizeof(int, 4)`, which will cause the `sizeof` expression to be evaluated as expected. A separate `-pp_sizeof` option must be used for each expression that will appear inside of a preprocessor `sizeof`.

## 19 Living with Lint

(or Don't Kill the Messenger)

*The comments in this chapter are suggestive and subjective. They are the thoughts and opinions of only one person and for this reason are written in the first person.*

When you first apply PC-lint Plus against a large C or C++ program that has not previously been linted, you will no doubt receive many more messages than you bargained for. You will perhaps feel as I felt when I first ran a Lint against a program of my own and saw how it rejected 'perfectly good' C code; I felt I wanted to write in C, not in Lint.

Stories of Lint's effectiveness, however, are legendary. PC-lint was, of course, passed through itself and a number of subtle errors were revealed (and continue to be revealed) in spite of exhaustive prior testing. I tested a public domain grep that I never dared use because it would mysteriously bomb. PC-lint found the problem - an uninitialized pointer.

It is not only necessary to test a program once but it should be continuously tested throughout a development/maintenance effort. Early in Lint's development we spent a considerable effort, over several days, trying to track down a bug that Lint would have detected easily. We learned our lesson and were never again tempted to debug code before linting.

But what do you do about the mountain of messages? Separating wheat from chaff can be odious especially if done on a continuing basis. The best thing to do is to adopt a policy (a policy that initially might be quite liberal) of what messages you're happy to live without. For example, you can inhibit all informational messages with the option `-w2`. Then work to correct only the issues associated with the messages that remain. DO NOT simply suppress all warnings with something like: `-e*` or `-w0` as this can disguise hard errors and make subsequent diagnosis very difficult. The policy can be automatically imposed by incorporating the error suppression options in a `.lnt` file (examples shown below) and it can gradually be strengthened as time or experience dictate.

Experience has shown that linting at full strength is best applied to new programs or new subroutines for old programs. The reasons for this is that the various decisions that a programmer has made are still fresh in mind and there is less hesitancy to change since there has been much less 'debugging investment' in the current design. Decisions such as, for example, which objects should be `signed` and which `unsigned`, can benefit from checking at full strength.

### 19.1 An Example of a Policy

An example of a set of practices with which I myself can comfortably live, is as follows.

Mistaking assignment for equality is a potential problem for C/C++. If a Boolean test is made of assignment as in

```
if( a = f( ) ) ...
```

PC-lint Plus will complain with message [720](#). If the assignment is wrapped with parentheses as in

```
if( (a = f()) ) ...
```

a different message ([820](#)) is used. This is done deliberately so that the programmer may distinguish between a conscious request and what appears to be accidental. Combining the assignment with testing is such a useful operation that I'm happy to put up with an extra pair of parentheses. Therefore, I suppress 820 with the option

```
-e820
```

At one time I mixed **unsigned** and **signed** quantities with almost reckless abandon. I now have considerably more respect for the subtle nuances of these two flavors of integer and now follow a more cautious approach. I had previously employed the options

```
-e713 -e737
```

(**713** involves assigning **unsigned** to **signed**, **737** is loss of sign). These inhibitions affect only some variation of assignment. We retain warnings about mixing **signed/unsigned** with binary operators.

I also no longer think it is a great idea to automatically inhibit **734** (sub-integer loss of precision). This message can catch all sorts of things such as assigning **int** to **short** when **int** is larger than **short**, assigning oversized **int** to **char**, assigning too large quantities into bit fields, etc.

I suppress messages about shifting **int** (and **long**) to the left but I want to be notified when they are shifted right as this can be machine-dependent and is generally regarded as a useless and hazardous activity. Therefore, I use **-e701 -e703**.

I want to run my code through at least two passes so that cross-functional checks can be made. The option is **-vt\_passes(2)**.

I place my list of favorite error-suppression options in a file called **options.lnt**. It looks like this:

```
options.lnt:
-e820          // parenthesized test of assignment
-e701 -e703    // shifting int left is OK
-vt_passes(2)  // use two value tracking passes
```

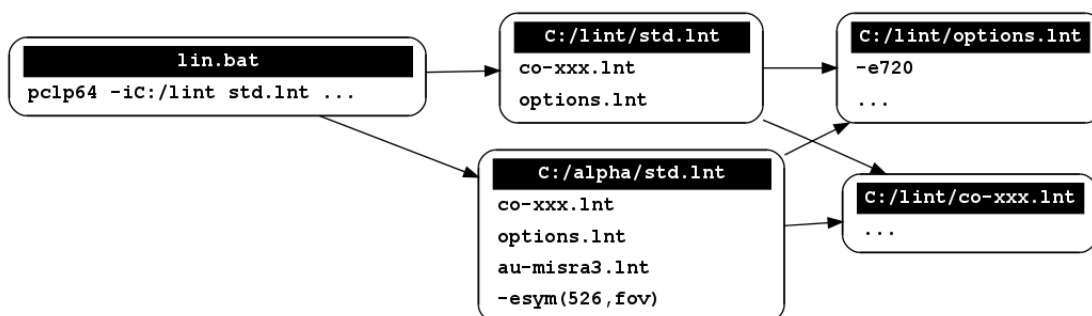
## 19.2 Recommended Setup

The recommended setup includes a default standard configuration (**std.lnt**) and a shell script (**lin.bat** or **lin.sh**) that invokes PC-lint Plus with **std.lnt**. The shell script might look like:

```
pclp64 -iC:/lint std.lnt ...
```

The "lin" script needs to be placed in your executable path.

When run from most directories, the file **std.lnt** is found in the PC-lint Plus directory (owing to the **-i** option appearing in the script). This in turn includes a compiler options file and a centralized **options.lnt** as shown below, also found in the PC-lint Plus directory. When run from a directory that has its own **std.lnt** file however, the local **std.lnt** overrides the standard one. In this way, each project can maintain its own configuration and running PC-lint Plus from a directory that doesn't have an explicit configuration will result in the default standard configuration being applied.



### 19.3 Final Thoughts

In summary, establish procedures whereby PC-lint Plus may be conveniently accessed for a variety of purposes. Lint small pieces of a project before doing the whole thing. Establish an error-message suppression policy that may initially be somewhat relaxed and can be strengthened in time. Lint full strength on new projects. But don't kill the messenger!

## 20 Common Issues

### 20.1 Syntax errors when using Boost

The Boost headers utilize certain compiler-specific quirks related to variadic macros whose behavior does not always match that of PC-lint Plus (or other compilers) resulting in syntax errors. The issue can be resolved by suppressing Boost's reliance of these quirks by adding the following options to your PC-lint Plus configuration:

```
--uBOOST_PP_VARIADICS
--uBOOST_PP_VARIADICS_MSVC
```

### 20.2 Suppressing MISRA/AUTOSAR/CERT C/CWE messages from library/system headers

By default, PC-lint Plus emits error, warning, and info messages from user code and only error messages from library code. The coding guideline author files (`au-misra2.lnt`, `au-misra3.lnt`, `au-autosar.lnt`, etc.) enable relevant messages for both user code and library code (including system headers). If it is not desired to check library code for coding guideline violations, the options `-wlib(4)` `-wlib(1)` can be placed immediately *after* the author file is referenced. This raises and immediately lowers the warning level for libraries resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.

### 20.3 Standard C++ keywords not recognized when using `au-misra-cpp.lnt`

MISRA C++ 2008 requires the use of C++03 and the `au-misra-cpp.lnt` file accordingly contains the option `-std(c++2003)` to set the C++ language version. In C++03 mode PC-lint Plus will not recognize keywords introduced in C++11 and later (including `constexpr`, `decltype`, `noexcept`, `nullptr`, and `static_assert`). To employ MISRA C++ compliance checking using a later version of C++, you will need to override the language version by adding the appropriate `-std` option (e.g. `-std=c++11`) anywhere after `au-misra-cpp.lnt` is referenced.

### 20.4 Running `pclp_config` from Windows produces errors or no result

Running `pclp_config` with the `--list` option should result in a list of supported compiler families. If running `pclp_config` in this way does not produce any output, the `pclp_config` utility probably isn't actually being invoked. If errors related to missing modules (i.e. `regex` or `pyyaml`) are produced, the modules either are not installed or are not installed for the version of Python being invoked. Depending on the version of Windows and the mechanism by which Python was installed, you may need to prefix the invocation of `pclp_config` with `python`, `py`, or the specific version of Python that is installed. Please contact support if you continue to encounter issues related to the use of `pclp_config`.

### 20.5 Resolving error 309 (`#error` directive encountered)

Error 309 is issued when a `#error` directive is reached while processing a header or module. This is a fatal error and PC-lint Plus will terminate after issuing the error. `#error` directives typically appear in header files to diagnose error conditions at preprocessing time. Almost all instances of this message are due to missing or incorrectly defined macros. The location provided with this message is where the `#error` directive appears and inspection of the surrounding source code should reveal the condition for which the directive can be avoided. A typical example will look something like:

```
#ifndef SKIP_FEATURE
#if VERSION < 5
#error "Feature requires version 5 or greater"
#endif
#endif
```



The `#error` directive above will be reached if `SKIP_FEATURE` is not defined and `VERSION` is either not defined or is defined with a value less than 5. The `#error` directive can be avoided either by defining the macro `SKIP_FEATURE` or defining `VERSION` with a value of 5 or greater. After determining the appropriate macro and value, the `-d` option can be used to define the macro appropriately for PC-lint Plus, e.g.:

```
-dSKIP_FEATURE
```

will define the `SKIP_FEATURE` macro and:

```
-dVERSION=7
```

will define the `VERSION` macro with the value 7. The appropriate option should be added to your project configuration file. Another common theme looks like:

```
#if defined(ARCH_ARM)
...
#elif defined (ARCH_PIC)
...
#elif defined (ARCH_PCS)
...
#else
#error "Unknown architecture"
#endif
```

In this case the header expects a macro corresponding to the appropriate architecture to be defined. If none of the recognized macros are defined, the `#error` directive is reached. In this case, the solution is to use the `-d` option to define the correct architecture macro. Note that if you used `pclp_config.py` to generate a compiler configuration and a macro of this nature is missing, it may indicate that a necessary CPU target option used to compile your project was not provided to `pclp_config.py` through the `--compiler-options` option. If this is the case, it is preferable to regenerate the compiler configuration with the appropriate CPU target options rather than adding a macro definition manually.

## 20.6 Resolving error 330 (static assert failed)

Error 330 is issued when the condition of a static assertion evaluates to false. This is a fatal error and PC-lint Plus will terminate after issuing the message. Like `#error` directives, static assertions are employed to diagnose error conditions detected during compilation time. In most cases, a failed static assertion is due to incorrect type size configuration or a missing or incorrectly defined macro.

The location provided with this message is where the static assert appears and inspection of the corresponding predicate should reveal cause of the failure. For example:

```
static_assert(VERSION >= 5);
```

will fail if the `VERSION` macro is defined with a value less than 5 and:

```
static_assert(sizeof(long) == 8);
```

will fail if the size of the `long` type is not equal to 8.

In the first case, the macro can be defined with the appropriate value using the `-d` option, e.g. `-dVERSION=5`. In the second case, the scalar data [size options](#) can be used to inform PC-lint Plus of the correct type size of the target platform.

## 20.7 Resolving errors 305 (unable to open module) and 307 (unable to open indirect file)

Messages [305](#) and [307](#) are issued when PC-lint Plus fails to open the specified file. This can occur if the file does not exist or is not present in the locations PC-lint Plus is searching for it. Ensure that the file is correctly spelled. If the file does not exist in the directory from which PC-lint Plus is invoked, you will need

to either specify the path of the file or use a `-i` option to specify the directory containing the file so that PC-lint Plus can find it.

If the file name reported by PC-lint Plus does not match the provided filename, you may need to surround the filename in quotes to prevent the shell from interpreting space or other special characters, see the below issue for more information.

## 20.8 Syntax errors when using options that contain spaces

When providing arguments to PC-lint Plus on the command line, you will need to surround arguments that contain spaces with double quotes to prevent your command interpreter from treating the spaces as an argument separator. For example:

```
pclp -iC:/directory with spaces C:/lint/module with spaces.c
```

will produce an error from PC-lint Plus similar to:

```
<command line> 3 error 305: unable to open module 'with'
with
~
```

because the arguments provided to PC-lint Plus from the command interpreter are:

```
-iC:/directory
with
spaces
C:/lint/module
with
spaces.c
```

PC-lint Plus will process the first argument (`-iC:/directory`) as an include directory and then try to open a file named `with` and fail with error 305. The correct way to pass these arguments to PC-lint Plus is to surround the arguments with double quotes:

```
pclp64.exe "-iC:/directory with spaces" "C:/lint/module with spaces.c"
```

When appearing inside of an indirect (`.lnt`) file, *filenames* containing spaces should similarly be surrounded with double quotes. *Options*, such as `-i` options should be enclosed with double quotes with the first quote character appearing *after* the `-` or `+` that starts the argument, e.g.:

```
-"iC:/directory with spaces"
```

as arguments starting with a double quote are assumed to represent a file name, not an option. See also [Options on the Command Line](#) and [Quoted Options and Arguments](#).

## 20.9 Resolving error 322 (unable to open include file)

Message [322](#) is issued when PC-lint Plus fails to open a file specified by an `#include` directive. PC-lint Plus searches for header files in several places including search directories specified by `-i` options, any directories specified by the `INCLUDE` environment variable, the directory of the including file if the `fdi` flag option is enabled, and the directories of the current `#include` stack if the `fsi` option is enabled.

In most cases, the solution is to add a `-i` option with the directory of the specified file to cause PC-lint Plus to search the correct directory. Note that if the `#include` directive contains a relative path, the directory specified by the `-i` option must be the location from which this relative path can be accessed. For example, if the `#include` directive looks like this:

```
#include "inc/header.h"
```

and the full path of the `header.h` file is `C:/sys/inc/header.h`, the correct `-i` option would be `-iC:/sys`, not `-iC:/sys/inc`.

## 20.10 Syntax errors when using options containing parentheses on the command prompt

Parentheses have special meaning in some command interpreters and need to be quoted to suppress their meaning. For example, invoking PC-lint Plus from the command line as:

```
pclp -os(results.txt) test.c
```

may result in an error reported by your command interpreter as:

```
syntax error near unexpected token `('
```

There are several possible solutions. You can quote the argument to suppress the special behavior of the parentheses:

```
pclp "-os(results.txt)" test.c
```

In most cases PC-lint Plus will accept square brackets in place of parentheses within options. Since most command interpreters do not attribute special behavior to square brackets, the command can be written as:

```
pclp -os[results.txt] test.c
```

although quotes will likely be necessary if your option contains space characters. Finally, you can place the option in an indirect (.lnt) file.

## 21 Frequently Asked Questions

### 21.1 How do I keep PC-lint Plus from processing header files?

Header files contain declarations and definitions that are critical to the understanding of the program that uses them. PC-lint Plus needs to be able to properly parse your header files in order to produce accurate analysis. If you are receiving error messages from headers (as opposed to warnings or infos), this represents a configuration issue that must be addressed. If you are receiving warnings or infos from library headers when utilizing a coding guidelines configuration file (e.g. MISRA), see [Suppressing MISRA/AUTOSAR/CERT C/CWE messages from library/system headers](#).

### 21.2 When do I need to use `imposter`?

The provided `imposter` program is only used during the generation of a project configuration. For projects with a small number of source files that are not compiled with different compiler options, it is often sufficient for a project configuration to simply consist of the names of the modules that PC-lint Plus should analyze. For projects with many modules or modules that are compiled with different options, it is usually easier to automatically generate a project configuration by extracting information from the build process. If your build system supports the generation of a JSON compilation database, you can use `pclp_config` to generate a [project configuration using this database](#), without the need for `imposter`. Otherwise, you can build your project using the `imposter` program as a replacement for your compiler, the result of which will be a YAML-formatted log containing compiler invocations which can then be used by `pclp_config` to generate a project configuration.

### 21.3 How should `imposter` be compiled?

The `imposter` program is distributed as a single C source file and can be compiled with any C compiler. As the `imposter` program is run as a drop-in replacement for your compiler during the build process, it should be compiled as an executable that will run in your host environment. For example, if you are building for an embedded device on Windows, you should compile `imposter` as a Windows executable, not an executable for your target device.

### 21.4 What can I do to improve analysis speed?

The performance profile of PC-lint Plus will depend on a variety of factors including the characteristics of the project being analyzed, the options used, and the hardware available. By default, PC-lint Plus analyzes modules one at a time. If the machine running PC-lint Plus has multiple processor cores, the `-max_threads` option can be used to improve performance by analyzing multiple modules at a time, e.g. `-max_threads=8` will allow PC-lint Plus to analyze up to 8 modules at a time which will result in significantly faster analysis if the machine has 8 or more processing cores. Some projects may also benefit from the use of [precompiled headers](#).

### 21.5 PC-lint Plus has crashed, what do I do next?

The most common cause of a crash of PC-lint Plus is memory exhaustion which can be exacerbated by the use of `-max_threads` option which increases memory consumption. When using the `-max_threads` option, crashes from memory exhaustion may be inconsistent. To determine if the crash is memory related to memory exhaustion, a memory profiler can be used to determine how much memory was being used by PC-lint Plus at the point of the crash. If the `-max_threads` option is set to a large value, lowering this value may mitigate the crash. If PC-lint Plus is running on a machine employing antivirus software, you may want to try running PC-lint Plus with the antivirus software temporarily disabled as some antivirus software has the potential to interfere with the operation of PC-lint Plus.

If memory exhaustion has been ruled out as a possible culprit, you should contact Vector Informatik GmbH support for further guidance. In most cases, a reproducible test case will be needed. You should also run

with the PC-lint Plus debug binary and provide any internal error information produced before the crash if applicable.

## 21.6 Why isn't my option working?

There are several possible reasons a suppression (or other option) may not work as expected including:

- The option is incorrect. For example, `-620` is being used instead of the correct `-e620`.
- Lint options are processed **in order**. Make sure that your modules are specified *after* any appropriate message suppressions or option (`.lnt`) files. Suppression options need to be processed before the corresponding module to be effective. A `-os` option takes effect from the time it is encountered so if it appears at the end of the command it will not be very effective.
- If the option appears in a lint comment, make sure that the word `lint` appears at the beginning of the comment with no spaces at the beginning. For example:

```
//lint -e620      OK
// lint -e620     Won't work
//-e620           Won't work
```

- An inappropriate option is being used to attempt to enable or suppress a message. For example, a `-esym` option is being used with a message that does not have a *Symbol* parameter. The `+paraminfo` option will cause the specified messages to be prefixed with information about the type of content of each parameter in the message which can be used to determine the appropriate parameterized suppression option to employ.

## 21.7 How can I temporarily disable suppressions of one or more messages?

The `++efreeze` option will cause any following suppression options to be ignored. When used by itself, this option will affect all messages, one or more message numbers may be specified as an argument to this option to limit the behavior to the specified messages. The `+efreeze` option is similar but can be later reverted using the `-efreeze` option allowing for the effect to be limited to particular files or code regions. See the description of these options for more information.

## 21.8 How can I customize existing messages or add new messages?

Existing messages may be customized by adding text to the message when it is emitted using the `-append` option. This option is used extensively in the coding guideline configuration files (e.g. `au-misra3.lnt`) to include references to the related coding guidelines when issuing messages.

New checks and messages may be created using the [Queries](#) feature. Significant use of this feature is employed in the `au-barr.lnt` file to create messages to support the BARR-C:2018 guidelines. This configuration file is included in the `lnt` directory of the PC-lint Plus distribution package.

## 21.9 How can I change the message category associated with a message?

The message category associated with every message ("error", "warning", etc.) cannot be changed. If the goal is to convey a heightened sense of severity for specific messages, `-append` option may be used to cause custom text to be appended to the message when it is issued. If the goal is to elicit a build failure by causing PC-lint Plus to terminate with a non-zero exit code when particular messages have been issued, the `-zero_err` option can be used. See the description of this option for more information. If the goal is to cause the output to conform to the format expected by an IDE integration (which may recognize errors and warnings but have no concept of elective notes), see the `-format_category` option.

## 21.10 How can I generate XML/HTML output?

The `env-xml.lnt` and `env-html.lnt` files included in the `lnt` directory of the PC-lint Plus distribution package can be used to emit XML or HTML output. These files may be modified to customize the output for your specific needs.

## 21.11 Can PC-lint Plus generate custom reports?

Aside from the summary report provided by the `-summary` option, [stack usage reports](#), and [thread analysis reporting](#), PC-lint Plus does not produce reports. The output of PC-lint Plus is very customizable (see [Message Presentation](#)) in order to facilitate the use of custom scripts that can generate reports during a post-processing step.

## 21.12 Where can I find `msg.txt` / `msg.json`?

The `msg.txt` and `msg.json` files containing a list of all supported PC-lint Plus messages in plain text or JSON format are no longer included in the distribution package. These files may be dynamically generated by PC-lint Plus by invoking it as:

```
pclp -dump_messages[file=msg.txt]
```

or:

```
pclp -dump_messages[file=msg.json,format=json]
```

See the description of the `-dump_messages` option for other supported formats and relevant information.

## 21.13 How can I apply options only to C files but not C++ files (or vice versa)?

Use the `-header` option, as in:

```
-header(lint.h)
```

and then within the specified header (`lint.h` in this example) add the following:

```
#ifdef __cplusplus
    //lint <cpp-options>
#else
    //lint <c-options>
#endif
```

where `<cpp-options>` and `<c-options>` correspond to the options to be processed for C++ and C modules, respectively. The `-header` option causes the specified file to be automatically included at the beginning of every analyzed module. The `__cplusplus` macro is defined by PC-lint Plus at the beginning of every C++ module and can be used to conditionally include source code, including lint comments, based on the current language mode.

## 21.14 Can PC-lint Plus recursively scan sub directories for the source code?

No, but you can obtain a list of files from a command prompt. For example:

```
dir /s/b *.cpp > project.lnt
```

will recursively scan the current directory for C++ source files on Windows and:

```
find . -name '*.cpp' > project.lnt
```

will accomplish the same on Linux or macOS. You can then use `project.lnt` as an argument to PC-lint Plus. You confirm that all those files are in the same project.

## 22 Messages

Every diagnostic message has an associated message number. By looking up the number in the list below you can obtain additional information about the cause of the diagnostic. This information can also be extracted from PC-lint Plus as a plain text or JSON file using the `-dump_messages` option.

Messages pertaining specifically to C++ generally reside in the 1xxx and 3xxx ranges while those applicable to C or both C and C++ reside in the xxx and 2xxx ranges. After a possible 1000 is subtracted off, the remainder lies in the range 0-999.

Remainders in the range 1-399 are errors that may be the result of incorrect syntax, a violation of the semantics of the language or a processing error. Some serious errors can be fatal, unsuppressible, or both. Errors are generally mistakes in the program or the PC-lint Plus configuration and should be corrected. While most errors can be suppressed, it is not advisable to do so unless the root cause is understood and it is determined that suppression is both safe and the only available course of action. Suppressing error messages can hide configuration issues that may result in incorrect or incomplete analysis.

Remainders in the 400-699 range are warning messages that indicate that something is potentially wrong with the program being examined. Remainders in the range 700-899 designate informational messages; these messages serve to point out unusual constructs, generally acknowledged bad practices, and potential pitfalls.

Remainders in the range 900-999 are called "Elective Notes" and diagnose specific points of interest that do not necessarily represent any deficiency in the code but may be of interest to certain users. They are not automatically presented. You may examine the list to see if you wish to be alerted to any of them.

Aside from the 4 ranges mentioned above, the 4xxx, 5xxx, and 6xxx ranges are used for C and C++ error messages, the 8xxx range is reserved for user-defined messages, and the 9xxx ranges is used for C and C++ elective notes. Messages in the 7xxx range are reserved for future use.

Note that there are over 2000 clang compiler errors that PC-lint Plus reports in the range 4001-6999. Because these messages are generally self-explanatory error messages, they are not included in this document.

CERT C guideline titles are excerpted from [SEI CERT C Coding Standard](#) © Carnegie Mellon University 1995–2016. See chapter [25](#) for full details.

| Range     | Description         | Warning Level |
|-----------|---------------------|---------------|
| 1-199     | C Syntax Errors     | 1             |
| 200-299   | Internal Errors     | 1             |
| 300-399   | Fatal Errors        | 1             |
| 400-699   | C Warnings          | 2             |
| 700-899   | C Informational     | 3             |
| 900-999   | C Elective Notes    | 4             |
| 1000-1199 | C++ Syntax Errors   | 1             |
| 1200-1299 | Internal Errors     | 1             |
| 1300-1399 | C++ Fatal Errors    | 1             |
| 1400-1699 | C++ Warnings        | 2             |
| 1700-1899 | C++ Informational   | 3             |
| 1900-1999 | C++ Elective Notes  | 4             |
| 2000-2199 | C Syntax Errors     | 1             |
| 2200-2399 | Reserved            |               |
| 2400-2699 | C Warnings          | 2             |
| 2700-2899 | C Informational     | 3             |
| 2900-2999 | C Elective Notes    | 4             |
| 3000-3199 | C++ Syntax Errors   | 1             |
| 3200-3399 | Reserved            |               |
| 3400-3699 | C++ Warnings        | 2             |
| 3700-3899 | C++ Informational   | 3             |
| 3900-3999 | C++ Elective Notes  | 4             |
| 4000-6999 | C and C++ Errors    | 1             |
| 7000-7999 | Reserved            |               |
| 8000-8999 | User Defined        | 3             |
| 9000-9999 | Misc Elective Notes | 4             |

## Glossary

A few of the terms used in the commentary below are:

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>argument</i>    | The actual argument of a function as opposed to a dummy (or formal) parameter of a function (see <i>parameter</i> below).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>arithmetic</i>  | Any of the integral types (see below) plus <code>float</code> , <code>double</code> , and <code>long double</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>Boolean</i>     | In general, the word Boolean refers to quantities that can be either true or false. An expression is said to be Boolean (perhaps it would be better to say ‘definitely Boolean’) if it is of the form: <code>operand op operand</code> where <code>op</code> is a relational ( <code>&gt;</code> <code>&gt;=</code> <code>&lt;</code> <code>&lt;=</code> ), an equality operator ( <code>==</code> <code>!=</code> ), logical <i>And</i> ( <code>&amp;&amp;</code> ) or logical <i>Or</i> ( <code>  </code> ). A context is said to require a Boolean if it is used in an <code>if</code> or <code>while</code> clause or if it is the 2nd expression of a <code>for</code> clause or if it is an argument to one of the operators: <code>&amp;&amp;</code> or <code>  </code> . An expression needn’t be definitely Boolean to be acceptable in a context that requires a Boolean. Any integer or pointer is acceptable. |
| <i>declaration</i> | Gives properties about an object or function (as opposed to a definition).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <i>definition</i>  | that which allocates space for an object or function (as opposed to a declaration) and that may also indicate properties about the object. There should be only one definition for an object but there may be many declarations.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <i>integral</i>    | a type that has properties similar to integers. These include <code>char</code> , <code>short</code> , <code>int</code> , and <code>long</code> and the <code>unsigned</code> variations of any of these.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>scalar</i>      | any of the arithmetic types plus pointers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |



|                  |                                                                                                                                                                                                                                                     |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>lvalue</i>    | is an expression that can be used on the Left hand side of an assignment operator (=). Some contexts require lvalues such as autoincrement (++) and autodecrement (--).                                                                             |
| <i>macro</i>     | an abbreviation defined by a <code>#define</code> statement. It may or may not have arguments.                                                                                                                                                      |
| <i>member</i>    | elements of a <code>struct</code> and of a <code>union</code> are called members.                                                                                                                                                                   |
| <i>module</i>    | that which is compiled by a compiler in a single independent compilation. It typically includes all the text of a <code>.c</code> (or a <code>.cpp</code> or <code>.cxx</code> , etc.) file plus any text within any <code>#include</code> file(s). |
| <i>parameter</i> | A formal parameter of a function as opposed to an actual argument (see <code>argument</code> above).                                                                                                                                                |

## 22.1 Message Parameters

Most messages are parameterized with one or more pieces of information that may be different each time the message is issued such as the name of a symbol being referenced, the types involved in a conversion, or a string representing dynamic text. These parameters can be employed for suppression purposes using `-esym`, `-estring`, and `-etype`.

There are three categories of message parameters: *Strings*, *Symbols*, and *Types*. *Symbol* parameters in a message are represented as *symbol* in the message descriptions below and may be used with the `-esym` option to suppress the message. *Type* parameters are represented as *type* and may be used with `-etype` to suppress the message. `-etype` may also be used to suppress *symbol* parameters by using the type of a symbol appearing in the message (see `+typename` for information about obtaining a type string suitable for use with `-etype` in this fashion). *String* parameters consist of virtually every other italicized parameter including:

- context
- detail
- file
- integer
- name
- operator
- string
- strong-type

Additionally, some messages contain dynamic text, which is represented by a slash in the message such as:

444: `for statement condition tests incremented/decremented pointer for null`

`-estring` can be used to suppress such messages when one of these values is used in the message, e.g. `-estring(444, incremented)`. `-esym` can also be used to suppress on *String* parameters when the `fsn` flag is ON (which it is by default).

Detailed parameter information for specific instances of messages can be obtained by using `+paraminfo`.

## 22.2 Messages 1-999

### 1 unclosed comment

**error** End of file was reached with an open comment still unclosed.

**Supports CERT C MSC04-C - Use comments consistently and in a readable fashion**

**Supports MISRA C 2004 Rule 1.2**

### 2 unclosed quote

**error** An end of line was reached and a matching quote character (single or double) to an earlier quote character on the same line was not found.

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2004 Rule 1.2**

**3 *string* without #if**

**error** A #elif was encountered not in the scope of a #if, #ifdef or #ifndef.

**5 too many #endif directives**

**error** A #endif was encountered not in the scope of a #if or #ifdef or #ifndef.

**6 #include nested too deeply**

**error** The maximum internal #include nesting depth was reached, likely as the result of a #include cycle. This is a fatal error.

**8 unclosed #if**

**error** A #if (or #ifdef or #ifndef) was encountered without a corresponding #endif.

Supports AUTOSAR17 Rule M16-1-2

Supports AUTOSAR19 Rule M16-1-2

Supports MISRA C 2012 Rule 20.14

Supports MISRA C++ Rule 16-1-2

Supports MISRA C 2004 Rule 19.17

**9 *string* after #else**

**error** A given #if contained a #else, which in turn was followed by either another #else or a #elif. The error message gives the line of the #if statement that started the conditional that contained the aberration.

**10 expecting *detail***

**error** The expected token (provided in *detail*) could not be found. For example:

```
void foo(int, int);
void bar() {
    foo(1, 2;
}
```

will yield the error:

```
error 10: expecting ')'
    foo(1, 2;
           ^
supplemental 891: to match this '('
    foo(1, 2;
           ^
```

due to the missing closing parenthesis at the end of the function call.

**12 need < or "**

**error** After a #include is detected and after macro substitution is performed, a file specification of the form <filename> or "filename" is expected.

Supports MISRA C 2012 Rule 20.3

Supports MISRA C++ Rule 16-2-6

Supports MISRA C 2004 Rule 19.3

**13 'string' cannot be signed or unsigned**

**error** A type adjective such as `long`, `unsigned`, etc. cannot be applied to the type, which follows.

**15 symbol *symbol* redeclared (*type* vs. *type*)**

**error** The named symbol has been previously declared or defined in some other module (location given) with a type different from the type given by the declaration at the current location. The parenthesized type parameters provide the two differing types.

**Supports AUTOSAR17 Rule M3-2-2**

**Supports AUTOSAR19 Rule M3-2-2**

**Supports MISRA C 2012 Rule 8.4**

**Supports MISRA C++ Rule 3-2-2**

**Supports MISRA C++ Rule 3-2-4**

**Supports MISRA C 2004 Rule 8.4**

**16 unknown preprocessor directive**

**error** A `#` directive is not followed by a recognizable word. If this is not an error, use the `+ppw` option. (Section [4.12 Compiler Adaptation](#)).

**Supports AUTOSAR17 Rule M16-0-8**

**Supports AUTOSAR19 Rule M16-0-8**

**Supports MISRA C 2012 Rule 20.13**

**Supports MISRA C++ Rule 16-0-8**

**Supports MISRA C 2004 Rule 19.16**

**18 symbol *symbol* redeclared (*typedef*)**

**error** A symbol is being redeclared. The parameter *typedef* provides further information on how the types differ.

**Supports AUTOSAR17 Rule M2-10-6**

**Supports AUTOSAR17 Rule M3-2-1**

**Supports AUTOSAR19 Rule M3-2-1**

**Supports CERT C DCL40-C - Do not create incompatible declarations of the same function or object**

**Supports MISRA C 2012 Rule 8.2**

**Supports MISRA C 2012 Rule 8.3**

**Supports MISRA C++ Rule 2-10-6**

**Supports MISRA C++ Rule 3-2-1**

**Supports MISRA C 2004 Rule 8.4**

**19 declaration requires a type specifier**

**error** A name appeared by itself without an associated type in what appears to be a declaration, e.g.:

```
x;
```

In C, this is a declaration of `x` with an implicit `int` type and warning [601](#) would be issued. In C++ this is illegal as the type must be explicit.

**21 array initializer must be an initializer list**

**error** An initializer for an indefinite size array must begin with a left brace.

**24 expected an expression**

**error** An operator was found at the start of an expression but it was not a unary operator.

- 25 character constant too long for its type**  
**error** Too many characters were encountered in a character constant (a constant bounded by ' marks).
- 29 duplicated type-specifier, '*detail*'**  
**error** This message is issued in C90 mode when a type specifier is duplicated within a declaration. For example:
- ```
const const int i = 0;
```
- will result in message 29 when in C90 mode, which forbids duplicate specifiers. In C99 and later, duplicate specifiers are ignored and such a construct will instead be greeted with warning [2435](#).
- 31 redefinition of symbol *symbol***  
**error** A data object or function previously defined in this module is being redefined.  
 Supports AUTOSAR17 Rule M3-2-1  
 Supports AUTOSAR17 Rule M3-2-2  
 Supports AUTOSAR19 Rule M3-2-1  
 Supports AUTOSAR19 Rule M3-2-2  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C++ Rule 3-2-1  
 Supports MISRA C++ Rule 3-2-2  
 Supports MISRA C++ Rule 3-2-4  
 Supports MISRA C 2004 Rule 1.2
- 32 field size (member *symbol*) should not be zero**  
**error** The length of a field was given as non-positive, (0 or negative).
- 33 illegal constant '*integer*' in octal constant**  
**error** A constant was badly formed as when an octal constant contains one of the digits 8 or 9.
- 34 non-compile-time-constant initializer**  
**error** A non-constant initializer was found for a static data item.
- 35 initializer has side-effects**  
**error** An initializer with side effects was found for a static data item.
- 40 undeclared identifier *detail***  
**error** Within an expression, an identifier was encountered that had not previously been declared and was not followed by a left parenthesis.  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2004 Rule 1.2
- 44 need a switch for case-label**  
**error** A `case` or `default` statement occurred outside a switch.  
 Supports MISRA C 2012 Rule 16.2  
 Supports MISRA C 2004 Rule 15.1

**47 invalid argument type *type* to unary expression**

**error** Unary minus requires an arithmetic operand.

**48 indirection requires pointer operand (*type* invalid)**

**error** Unary `*` or the left hand side of the ptr (`->`) operator requires a pointer operand.  
Supports MISRA C 2012 Rule 10.1

**50 attempted to take the address of a non-lvalue of type *type***

**error** Unary `&` operator requires an lvalue (a value suitable for placement on the left hand side of an assignment operator).

**51 expected integral type for bitwise complement operator**

**error** Unary `~` expects an integral type (signed or unsigned char, short, int, or long).

**52 expected an lvalue**

**error** The autodecrement (`--`) and autoincrement (`++`) operators require an lvalue (a value suitable for placement on the left hand side of an assignment operator). Remember that casts do not normally produce lvalues. Thus

```
++(char *)p;
```

is illegal according to the ANSI/ISO standard. This construct is allowed by some compilers and is allowed if you use the `+fpc` option (Pointer Casts are lvalues). (See Section [4.11 Flag Options](#))

**53 expected a scalar and not expression of type *type***

**error** Autodecrement (`--`) and autoincrement (`++`) operators may be applied only to scalars (arithmetics and pointers) or to objects for which these operators have been defined.

**54 *division/remainder* by 0**

**error** The constant 0 was used on the right hand side of the division operator (`/`) or the remainder operator (`%`).  
Supports MISRA C 2012 Rule 1.3  
Supports MISRA C 2004 Rule 1.2  
Supports CWE-369 - Divide By Zero

**56 pointer addition requires an integral type**

**error** Add/subtract operator requires scalar types and pointers may not be added to pointers.

**57 operands to bitwise operator must be integral**

**error** Bit operators (`&`, `|` and `^`) require integral arguments.

**59 number of bits to shift by must be an integer**

**error** The amount by which an item can be shifted must be integral.

**60 value to be shifted must be an integer**

**error** The value to be shifted must be integral.

**66 'void' must be the first and only parameter if specified**

**error** A void type was employed where it is not permitted. If a void type is placed in a prototype then it must be the only type within a prototype.

**72 bad option '*option*': *detail***

**error** An unrecognized or invalid option was encountered on the command line, in the LINT environment variable, in an indirect file, or in a lint comment. The *option* parameter contains the offending option and *detail* provides additional information about the issue.

**76 cannot open file '*file*': *detail***

**error** *file* is the name of the file. The named file could not be opened for output. *detail* contains information about the failure. This error is issued when PC-lint Plus is directed to write to a file with the options `-oe/+oe`, `-os/+os`, `-write_file`, or `+stack` but it unable to open the specified file for writing.

**82 *string detail* must not return void expression**

**error** The expression form of a `return` statement was used in a function that was declared as returning `void`, e.g.:

```
void foo() {
    return 1; // Error 82
}
```

Note that the ANSI/ISO C standard does not allow the expression form of a `return` statement in a function that returns `void`, even when the provided expression has `void` type. For example:

```
void foo_v();
int foo_i();

void bar(int x) {
    if (x) return foo_v();           // Error 82
    else return (void) foo_i();      // Error 82
}
```

If your compiler supports the latter case, you may safely suppress this message in such situations.

**83 incompatible pointer types (*type* and *type*) with subtraction**

**error** Two pointers being subtracted have indirect types that differ.

**85 array *symbol* has dimension 0**

**error** An array (named *symbol*) was declared without a dimension in a context that required a non-zero dimension.

**86 structure *symbol* has zero elements**

**error** A structure was declared (in a C module) that had no data members. Though legal in C++ this is not legal C.  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2004 Rule 1.2

- 92 negative length of *integer* for bit field *string***  
**error** A negative array dimension or bit field length is not permitted.
- 95 expected a macro parameter**  
**error** The **#** operator was found within a macro definition but was not immediately followed by a parameter of the macro as is required by the standards.
- 104 cannot combine with previous '*parameter*' declaration specifier**  
**error** Two consecutive conflicting types were found such as **int** followed by **double**. Remove one of the types.
- 106 illegal constant**  
**error** A string constant was found within a preprocessor expression as in  

```
#if ABC == "abc"
```

  
Such expressions should be integral expressions.
- 107 label *name* not defined**  
**error** The *name* appeared in a **goto** but there was no corresponding label.  
**Supports AUTOSAR17 Rule M6-6-2**  
**Supports AUTOSAR19 Rule M6-6-2**  
**Supports MISRA C++ Rule 6-6-2**
- 108 invalid context for '**break**' statement**  
**error** A **continue** or **break** statement was encountered without an appropriate surrounding context such as a **for**, **while**, or **do loop** or, for the **break** statement only, a surrounding **switch** statement.
- 115 struct/union not defined**  
**error** A reference to a structure or a **union** was made that required a definition and there is no definition in scope. For example, a reference to **p->a** where **p** is a pointer to a **struct** that had not yet been defined in the current module.  
**Supports MISRA C 2004 Rule 18.1**
- 116 inappropriate storage class**  
**error** A storage class other than **register** was given in a section of code that is dedicated to declaring parameters. The section is that part of a function preceding the first left brace.
- 117 inappropriate storage class**  
**error** A storage class (indicated as either **auto** or **register**) was provided outside any function. Such storage classes are appropriate only within functions.
- 118 too few arguments (*integer* vs *integer*) for prototype**  
**error** The number of arguments provided for a function was less than the number indicated by a prototype in scope.  
**Supports MISRA C 2004 Rule 16.6**  
**Supports CWE-685 - Function Call With Incorrect Number of Arguments**

- 119 too many arguments (*integer* vs *integer*) for prototype**  
**error** The number of arguments provided for a function was greater than the number indicated by a prototype in scope.  
**Supports MISRA C 2004 Rule 16.6**  
**Supports CWE-685 - Function Call With Incorrect Number of Arguments**
- 124 pointer to void not allowed**  
**error** A pointer to `void` was used in a context that does not permit `void`. This includes subtraction, addition and the relationals (`>` `>=` `<` `<=`).
- 128 pointer to function not allowed**  
**error** A pointer to a function was found in an arithmetic context such as subtraction, addition, or one of the relationals (`>` `>=` `<` `<=`).
- 130 *type* is not an integral type**  
**error** The expression in a `switch` statement must be some variation of an `int` (possibly `long` or `unsigned`) or an `enum`.
- 131 too few arguments provided to function-like macro invocation**  
**error** This message is issued when a macro with arguments (function-like macro) is invoked and an incorrect number of arguments is provided.  
**Supports MISRA C 2004 Rule 19.8**
- 132 expected function definition**  
**error** A function declaration with identifiers between parentheses is the start of an old-style function definition (K&R style). This is normally followed by optional declarations and a left brace to signal the start of the function body. Either replace the identifier(s) with type(s) or complete the function with a function body.
- 136 illegal macro name**  
**error** The ANSI/ISO standard restricts the use of certain names as macros. `defined` is on the restricted list.  
**Supports MISRA C 2012 Rule 1.3**  
**Supports MISRA C 2012 Rule 21.1**  
**Supports MISRA C 2004 Rule 1.2**
- 138 cannot create recursive relationship between '*strong-type*' and '*strong-type*'**  
**error** An attempt was made to add a strong type `parent` to a `typedef` type. The attempt is either explicit (with the `-strong` option) or implicit with the use of a `typedef` to a known strong type. This attempt would have caused a loop in the strong parent relationship. Such loops are simply not tolerated.
- 139 cannot take sizeof a function**  
**error** There is an attempt to take the `sizeof` a function.
- 148 member *name* previously declared**  
**error** The indicated member was previously declared within the same structure or union. Although a redeclaration



of a function may appear benign it is just not permitted by the rules of the language. One of the declarations should be removed.

**157 no data may follow an incomplete array**

**error** An incomplete array is allowed within a struct of a C99 or C++ program but no data is allowed to appear after this array. For example

```
struct A { int x; int a[]; int b; };
```

This diagnostic is issued when the 'b' is seen.

**160 the sequence ({ is non standard and is taken to introduce a GNU statement expression**

**error** PC-lint Plus encountered the sequence '{' in a context where an expression (possibly a sub-expression) is expected. For example:

```
int n = ({ //Error 160 here
    int y = foo ();
    int z;
    if (y > 0)
        z = y;
    else z = - y;
    z; })
// Now n has the last value of z.
```

In addition to being a non-standard GNU extension, there are some caveats described in the GCC documentation (especially when used in C++) that can lead to subtle bugs. Programmers who intend to work only with C code with the GNU extensions may safely disable this diagnostic.

**161 repeated use of parameter *symbol* in parameter list**

**error** The name of a function parameter was repeated. For example:

```
void f( int n, int m, int n ) {}
```

will cause this message to be issued. Names of parameters for a given function must all be different.

**175 cannot pass *string* to variadic *string*; expected type from format string was *type***

**error** An initializer list or an expression of a type that cannot be passed as a variadic function argument was given as the argument to a `printf/scanf` style function. The *string* parameter specifies the type of the argument passed, the *type* parameter specifies the type that was expected from the format string.

**Supports CERT C DCL11-C - Understand the type issues associated with variadic functions**

**176 operand of type *type* cannot be cast to function pointer type *type***

**error** An attempt was made to perform an illegal cast from a type (such as a float) to a function pointer for which such conversion is undefined.

**Supports MISRA C 2012 Rule 11.1**

**Supports MISRA C 2004 Rule 11.1**

**177 operand of type *type* cannot be cast to object pointer type *type***

**error** An attempt was made to perform an illegal cast from a type (such as a float) to an object pointer for which such conversion is undefined.

**Supports MISRA C 2012 Rule 11.7**

**Supports MISRA C 2004 Rule 11.2**

**178 function pointer of type *type* cannot be cast to type *type***  
**error** An attempt was made to perform an illegal cast from a function pointer to a type (such as a float) for which such conversion is undefined.  
 Supports MISRA C 2012 Rule 11.1  
 Supports MISRA C 2004 Rule 11.1

**179 object pointer of type *type* cannot be cast to type *type***  
**error** An attempt was made to perform an illegal cast from an object pointer to a type (such as a float) for which such conversion is undefined.  
 Supports MISRA C 2012 Rule 11.7  
 Supports MISRA C 2004 Rule 11.2

**180 cannot generate mangled name for symbol *symbol* whose type depends on *string***  
**error** Name mangling refers to the process of taking the declared name of a symbol and producing a new name which additionally incorporates information about the symbol, for example its type, return types, parameter types, enclosing classes or namespaces, etc. Typical compilers and linkers use mangled names to implement function overloading and to uniformly refer to and disambiguate external entities.

When certain combinations of compiler extensions are mixed in a manner that does not conform to the language standard, situations may arise where it is not possible to systematically construct a mangled name to allow an entity to be cross-referenced between translation units. This message is parameterized by the category of grammatical entity for which mangling failed.

**181 invalid direct member access of atomic structure or union of type *type***  
**error** A struct or union qualified with `_Atomic` was directly accessed via the `.` or `->` operator. For example:

```
typedef struct a {
    unsigned char i;
    unsigned char j;
} a_t;

_Atomic a_t myAtomStruct;

void foo() {
    myAtomStruct.i = 7U;           // error 181
    _Atomic a_t *atom_struct_ptr = &myAtomStruct;
    atom_struct_ptr->j = 42U;      // error 181
}
```

will produce this message. Operations on atomic variables are guaranteed to avoid data races, but only when done so through the dedicated access functions. Directly accessing members of an atomic structure or union results in undefined behavior.

Supports MISRA C 2012 AMD4 Rule 12.6

**305 unable to open module '*file*'**  
**error** The module specified in the *file* parameter could not be opened for reading, perhaps because it does not exist where PC-lint Plus expected to find it. If the module is not in the directory from which PC-lint Plus was invoked, it will be searched for in the directories specified by any `-i` options. If the file exists in a directory not being searched, a `-i` option can be used to cause PC-lint Plus to search the appropriate directory.

- 307 cannot open indirect file '*file*'**  
**error** The indirect file specified in the *file* parameter could not be opened for reading, perhaps because it does not exist where PC-lint Plus expected to find it. If the file is not in the directory from which PC-lint Plus was invoked, it will be searched for in the directories specified by any **-i** options. If the file exists in a directory not being searched, a **-i** option can be used to cause PC-lint Plus to search the appropriate directory.
- 308 can't write to file '*file*' for PCH construction**  
**error** stdout was found to equal NULL. This is most unusual.
- 309 #error detail**  
**error** The **#error** directive was encountered. This error is fatal by default, but can be bypassed using the **fce** flag.
- 311 indirect file depth limit of '*integer*' exceeded**  
**error** The indirect (*.lnt*) file nesting depth has reached an internal limit (the current limit is 100). The most likely cause of this error is unintended recursion between indirect files.
- 314 cannot use indirect file '*file*' again**  
**error** The indirect file named was previously encountered. If this was not an accident, you may suppress this message.
- 315 message limit exceeded**  
**error** The maximum number of messages specified using the **-limit** option was exceeded.
- 318 EOF for a module found within a macro argument list**  
**error** The end of a module was encountered while processing the argument list of a macro invocation.
- 319 size option misconfiguration: '*type*' has size *integer* and '*type*' has size *integer***  
**error** A fatal inconsistency in the sizes of the fundamental data types was introduced by use of the size options. The **-s** option allows for configuration of the sizes of the fundamental data types. If these options are used to specify sizes that violate the basic tenets of the language, this message will be issued. Such an example would include specifying a byte size for **short int** that is larger than **int**.
- 322 unable to open include file '*file*'**  
**error** *file* is the name of the include file, which could not be opened. Directory search is controlled by options: **-i**, **+fdi**, **+fsi** and the INCLUDE environment variable (See Section 18.2.1 INCLUDE Environment Variable). This is a suppressible fatal message. If option **-e322** is used processing will continue.
- 330 static\_assert failed*string***  
**error** This message is issued when the constant-expression of a static-assert-declaration (either C11's **\_Static\_assert** or C++11's **static\_assert**) evaluates to false. If PC-lint Plus issues this error message but the compiler does not, see whether the Lint configuration matches the compiler configuration: consider potential differences in pre-defined macros and include search options, and ensure that size options match the target machine. Differences in these configuration details could lead to differences in evaluation of the constant-expression.

- 331 file '*parameter*' has been modified since the *string* '*parameter*' was built: *string* changed**  
**error** Use of the specified precompiled header was requested but was found to contain a reference to a header file that has been updated since the precompiled header was built. Since the precompiled header may no longer accurately represent the state of the corresponding header file, PC-lint Plus will terminate. To resolve the issue either rebuild the precompiled header file or remove the option to use the pre-compiled header.
- 333 cannot open '*file*' for *string* in secure mode**  
**error** A 'forbidden' file was opened. Opening such a file is considered a security violation by a hosted implementation.
- 334 precompiled header failure: '*string*'; skipping this module; consider deleting the PCH, '*string*', and trying again**  
**error** This message is given when the precompiled header file is missing, older than the original file, created by a previous incompatible version of PC-lint Plus, created using a different target configuration, or another issue that prevents the file from being loaded. The PCH file will be skipped. If this error is encountered, the PCH file should be deleted and reconstituted using the current version of PC-lint Plus with the same options that will be used when loading the file.
- 336 source file is not valid UTF-8**  
**error** Source files are expected to be encoded as UTF-8 text or UTF-16 text. The provided source file was presumed to contain UTF-8 text but an invalid byte sequence was encountered.
- 337 unable to read file '*file*': *detail***  
**error** This message is given when the internal file information inside the precompiled header does not match its contents.
- 338 precompiled header error: *string*; skipping this module; consider examining include path options and trying again**  
**error** This error indicates that there was an inconsistency in the way the precompiled header file was created and the way it is being used that prevents it from being loaded. The details of the issue are provided in the message text.
- 339 precompiled header for '*string*' was not created due to errors**  
**error** Precompiled header file creation was requested but an error occurred during the processing of a precompiled header candidate file that rendered the corresponding AST information unsuitable for use in a precompiled header file.
- 340 query assertion condition '*string*' failed**  
**error**
- 365 command pipe error: *string***  
**error** An error has occurred while processing a request related to a command pipe program. The details of the error are specified in the message in '*string*'.
- 366 regex error: *string***  
**error** An invalid regular expression has been used with the `-cond` option. The specific error is provided in '*string*'.

- 367 maximum hook recursion depth (*integer* levels) reached**  
**error** A limit has been reached on the number of recursively executing hooks. The limit that was reached is specified by '*integer*'.
- 368 invalid conditional expression: *string***  
**error** An invalid conditional expression has been provided to the `-cond` option. The specific error is provided in the text of the message.
- 369 hook field error while processing '*string*': *string***  
**error** An invalid field name was provided in a hook field specifier or an AST walk action was attempted on a non-walkable hook field.
- 370 options executed within a module cannot invoke additional modules**  
**error** An attempt was made to process a new module from within the module being processed. For example, a lint comment might contain an `-indirect` option resulting in the processing of options from a `.lint` file. If this indirect file contains the name of a module to process, the module will not be opened and this message will be issued instead. Processing will then continue normally for the current module.
- 373 lint comments cannot appear after a `#include` directive on the same line**  
**error** A lint comment appeared on the same line as an `#include` directive. Such usage is not currently supported and the lint comment will be ignored. Either place the lint comment before the `#include` directive, on the next line, or inside the file being included.
- 374 C++ module import '*string*' not found**  
**error**
- 375 C++ module import '*string*' could not be loaded**  
**error**
- 398 fatal error (requested by option): '*string*'**  
**error** A suppressible fatal error was requested through the use of the option `-fatal_error`. See also [399](#).
- 399 fatal error (requested by option): '*string*'**  
**error** An unsuppressible fatal error was requested through the use of the option `+fatal_error`. See also [398](#).
- 401 symbol *symbol* not previously declared static**  
**warning** The indicated *symbol* declared `static` was previously declared without the `static` storage class. This is technically a violation of the ANSI/ISO standard. Some compilers will accept this situation without complaint and regard the *symbol* as `static`.  
 Supports AUTOSAR17 Rule M3-3-2  
 Supports AUTOSAR19 Rule M3-3-2  
 Supports CERT C DCL36-C - Do not declare an identifier with conflicting linkage classifications  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C++ Rule 3-3-2  
 Supports MISRA C 2004 Rule 1.2  
 Supports MISRA C 2004 Rule 8.11

- 402 static function/variable symbol not defined**  
**warning** The named *symbol* was declared as a **static** function in the current module and was referenced but was not defined (in the module).
- 404 definition of *type* starts in 'file' but ends in 'file'**  
**warning** A **struct** (or **union** or **enum**) definition was started within a header file but was not completed within the same header file.
- 407 inconsistent use of tag *symbol***  
**warning** A tag specified as a **union**, **struct** or **enum** was respecified as being one of the other two in the same module. For example:
- ```

    struct tag *p;
    union tag *q;

```
- will elicit this message.  
**Supports MISRA C 2012 Rule 5.7**
- 408 case expression type (*type*) differs from switch expression type (*type*)**  
**warning** The expression within a **case** does not agree exactly with the type within the **switch** expression. For example, an enumerated type is matched against an **int**.
- 409 integer base for subscript operator is suspicious**  
**warning** An expression of the form *i*[...] was encountered where *i* is an integral expression. This could be legitimate depending on the subscript operand. For example, if *i* is an **int** and *a* is an array then *i*[*a*] is legitimate but unusual. If this is your coding style, suppress this message.  
**Supports CERT C ARR00-C - Understand how arrays work**  
**Supports CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer**  
**Supports CWE-129 - Improper Validation of Array Index**
- 410 size\_t not what was expected from fzl and/or fzu, using *type***  
**warning** This warning is issued if you had previously attempted to set the type of **sizeof** by use of the options **+fzl**, **-fzl**, or **-fzu**, and a later **size\_t** declaration contradicts the setting. This usually means you are attempting to lint programs for another system using header files for your own system. If this is the case we suggest you create a directory housing header files for that foreign system, alter **size\_t** within that directory, and lint using that directory.
- 411 ptrdiff\_t not what was expected from fdl option, using *type***  
**warning** This warning is issued if you had previously attempted to set the type of pointer differences by use of the **fdl** option and a later **ptrdiff\_t** declaration contradicts the setting. See suggestion in Error Message [410](#).
- 413 likely use of null pointer *symbol***  
**warning** From information gleaned from earlier statements, it appears likely that a null pointer (a pointer whose value is 0) has been used in a context where null pointers are inappropriate. Information leading to this determination is provided as a series of supplemental messages. See also message [613](#).  
**Supports AUTOSAR19 Rule A5-3-2**  
**Supports CERT C EXP34-C - Do not dereference null pointers**

Supports CERT C ARR00-C - *Understand how arrays work*  
 Supports CERT C ARR30-C - *Do not form or use out-of-bounds pointers or array subscripts*  
 Supports CERT C MEM11-C - *Do not assume infinite heap space*  
 Supports CERT C API00-C - *Functions should validate their parameters*  
 Supports CERT C MSC19-C - *For functions that return an array, prefer returning an empty array over a null value*  
 Supports CERT C POS54-C - *Detect and handle POSIX library errors*  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2004 Rule 1.2  
 Supports CWE-119 - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
 Supports CWE-123 - *Write-what-where Condition*  
 Supports CWE-125 - *Out-of-bounds Read*  
 Supports CWE-126 - *Buffer Over-read*  
 Supports CWE-127 - *Buffer Under-read*  
 Supports CWE-129 - *Improper Validation of Array Index*  
 Supports CWE-252 - *Unchecked Return Value*  
 Supports CWE-253 - *Incorrect Check of Function Return Value*  
 Supports CWE-391 - *Unchecked Error Condition*  
 Supports CWE-476 - *NULL Pointer Dereference*  
 Supports CWE-690 - *Unchecked Return Value to NULL Pointer Dereference*  
 Supports CWE-786 - *Access of Memory Location Before Start of Buffer*  
 Supports CWE-787 - *Out-of-bounds Write*

**414** **possible division by zero**  
**warning** The second argument to either the division operator (/) or the modulus operator (%) may be zero. Information is taken from earlier statements including assignments, initialization and tests. See Chapter 8 [Value Tracking](#).

Supports AUTOSAR17 Rule A5-5-1  
 Supports AUTOSAR19 Rule A5-6-1  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2004 Rule 1.2  
 Supports CWE-369 - *Divide By Zero*

**415** **likely out of bounds pointer access: excess of *integer* byte(s)**  
**warning** An out-of-bounds pointer was likely accessed. The parameter *integer* gives some idea how far out of bounds the pointer may be, measured in bytes. For example:

```
int a[10];
a[10] = 0;
```

results in a message containing the phrase 'excess of 4 bytes' if the size of `int` is 4. See Chapter 8 [Value Tracking](#).

Supports AUTOSAR17 Rule M5-0-16  
 Supports AUTOSAR17 Rule A5-2-5  
 Supports AUTOSAR19 Rule M5-0-16  
 Supports AUTOSAR19 Rule A5-2-5  
 Supports CERT C ARR30-C - *Do not form or use out-of-bounds pointers or array subscripts*  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2012 Rule 18.1  
 Supports MISRA C++ Rule 5-0-16  
 Supports MISRA C 2004 Rule 1.2  
 Supports CWE-119 - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
 Supports CWE-123 - *Write-what-where Condition*  
 Supports CWE-125 - *Out-of-bounds Read*  
 Supports CWE-126 - *Buffer Over-read*  
 Supports CWE-127 - *Buffer Under-read*

Supports **CWE-129** - *Improper Validation of Array Index*

Supports **CWE-786** - *Access of Memory Location Before Start of Buffer*

Supports **CWE-787** - *Out-of-bounds Write*

- 416** **likely creating out-of-bounds pointer: excess of *integer* byte(s)**  
**warning** An out-of-bounds pointer was created. See message **415** for a description of the *integer* parameter. *integer* and *string*. For example:

```
int a[10];
...
f( a + 11 );
```

Here, an illicit pointer value is created and is flagged as such by PC-lint Plus. Note that the pointer `a+10` is not considered by PC-lint Plus to be the creation of an out-of-bounds pointer. This is because ANSI/ISO C explicitly allows pointing just beyond an array. Access through `a+10`, however, as in `*(a+10)` or the more familiar `a[10]`, would be considered erroneous but in that case message **415** would be issued. See Chapter 8 [Value Tracking](#).

Supports **AUTOSAR17 Rule M5-0-16**

Supports **AUTOSAR17 Rule A5-2-5**

Supports **AUTOSAR19 Rule M5-0-16**

Supports **AUTOSAR19 Rule A5-2-5**

Supports **CERT C EXP08-C** - *Ensure pointer arithmetic is used correctly*

Supports **CERT C ARR30-C** - *Do not form or use out-of-bounds pointers or array subscripts*

Supports **MISRA C 2012 Rule 1.3**

Supports **MISRA C 2012 Rule 18.1**

Supports **MISRA C++ Rule 5-0-16**

Supports **MISRA C 2004 Rule 1.2**

Supports **CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*

Supports **CWE-123** - *Write-what-where Condition*

Supports **CWE-125** - *Out-of-bounds Read*

Supports **CWE-126** - *Buffer Over-read*

Supports **CWE-127** - *Buffer Under-read*

Supports **CWE-129** - *Improper Validation of Array Index*

Supports **CWE-468** - *Incorrect Pointer Scaling*

Supports **CWE-786** - *Access of Memory Location Before Start of Buffer*

Supports **CWE-787** - *Out-of-bounds Write*

- 417** **integral constant '*string*' has precision *integer* which is longer than long long int**  
**warning** The longest possible integer is by default 8 bytes (see the `+fll` flag and then the `-s11#` option). An integral constant was found to be even larger than such a quantity. For example: `0xFFFF0000FFFF0000F` requires 68 bits and would by default elicit this message. *string* is the token in error, and *integer* is the binary precision.

- 418** **passing null pointer to function *symbol*, *context***  
**warning** A NULL pointer is being passed to a function identified by *symbol*. The argument in question is given by *context*. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option `-function` or `-sem`. See Section 9.1 [Function Mimicry \(-function\)](#) and Section 9.2.1 [Possible Semantics](#).

Supports **CERT C EXP34-C** - *Do not dereference null pointers*

Supports **CERT C MSC19-C** - *For functions that return an array, prefer returning an empty array over a null value*

Supports **CWE-476** - *NULL Pointer Dereference*

Supports **CWE-690** - *Unchecked Return Value to NULL Pointer Dereference*



- 419** **apparent data overrun for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**  
**warning** This message is for data transfer functions such as `memcpy`, `strcpy`, `fgets`, etc. when the size indicated by the first cited argument (or arguments) exceeds the size of the buffer area cited by the second. The message may also be issued for user functions via the `-function` option. See Section 9.1 Function Mimicry (`-function`) and Section 9.2.1 Possible Semantics.
- Supports CERT C ARR38-C** - *Guarantee that library functions do not form invalid pointers*  
**Supports CERT C MSC19-C** - *For functions that return an array, prefer returning an empty array over a null value*  
**Supports MISRA C 2012 AMD1 Rule 21.17**  
**Supports MISRA C 2012 AMD1 Rule 21.18**  
**Supports MISRA C 2012 Rule 1.3**  
**Supports MISRA C 2004 Rule 1.2**  
**Supports CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
**Supports CWE-121** - *Stack-based Buffer Overflow*  
**Supports CWE-123** - *Write-what-where Condition*  
**Supports CWE-125** - *Out-of-bounds Read*  
**Supports CWE-126** - *Buffer Over-read*  
**Supports CWE-129** - *Improper Validation of Array Index*  
**Supports CWE-805** - *Buffer Access with Incorrect Length Value*
- 420** **apparent access beyond array for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**  
**warning** This message is issued for several library functions (such as `fwrite`, `memcpy`, etc.) wherein there is an apparent attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified. The function is specified by *symbol* and the arguments are identified by argument number. See Section 9.1 Function Mimicry (`-function`) and Section 9.2.1 Possible Semantics.
- Supports CERT C ARR38-C** - *Guarantee that library functions do not form invalid pointers*  
**Supports CERT C MSC19-C** - *For functions that return an array, prefer returning an empty array over a null value*  
**Supports MISRA C 2012 AMD1 Rule 21.17**  
**Supports MISRA C 2012 AMD1 Rule 21.18**  
**Supports CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
**Supports CWE-121** - *Stack-based Buffer Overflow*  
**Supports CWE-123** - *Write-what-where Condition*  
**Supports CWE-125** - *Out-of-bounds Read*  
**Supports CWE-126** - *Buffer Over-read*  
**Supports CWE-127** - *Buffer Under-read*  
**Supports CWE-129** - *Improper Validation of Array Index*  
**Supports CWE-805** - *Buffer Access with Incorrect Length Value*
- 421** **caution – function *symbol* is considered dangerous**  
**warning** This message is issued (by default) for the built-in function `gets`. This function is considered dangerous because there is no mechanism to ensure that the buffer provided as first argument will not overflow. Numerous exploits and vulnerabilities are attributed to the `gets` function including the Morris worm, which exploited the use of the `gets` function in the fingered program of target machines. Through the `-function` option or the `dangerous` semantic (9.2.1 `dangerous`), the user may designate other functions as dangerous. See also `-deprecate`.
- Supports CERT C STR31-C** - *Guarantee that storage for strings has sufficient space for character data and the null terminator*  
**Supports CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
**Supports CWE-120** - *Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*  
**Supports CWE-121** - *Stack-based Buffer Overflow*  
**Supports CWE-122** - *Heap-based Buffer Overflow*  
**Supports CWE-123** - *Write-what-where Condition*

**Supports CWE-125** - *Out-of-bounds Read*

**Supports CWE-193** - *Off-by-one Error*

**Supports CWE-242** - *Use of Inherently Dangerous Function*

**Supports CWE-676** - *Use of Potentially Dangerous Function*

#### **422** **warning** **passing to function *symbol* a negative value (*integer*) *context***

An integral value that appears to be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (*symbol*), the questionable value (*integer*) and the argument number (*context*). The function may be a standard library function designed to accept only positive values such as `malloc` or `memcpy` (third argument), or may have been identified by the user as such through the `-function` or `-sem` options.

The parameter type may in fact be `unsigned`. Thus:

```
void *malloc(unsigned);
void f() {
    int n = -1;
    int *p;
    p = malloc(n);           // warning 422
}
```

will result in the warning indicated.

**Supports MISRA C 2012 AMD1 Rule 21.18**

**Supports CWE-687** - *Function Call With Incorrectly Specified Argument Value*

#### **423** **warning** **assignment to custodial pointer *symbol* likely creates memory leak**

An assignment was made to a pointer variable (designated by *symbol*), which appeared to already be holding the address of an allocated object that had not been freed. The allocation of memory that is not freed is considered a memory leak.

**Supports CWE-772** - *Missing Release of Resource after Effective Lifetime*

#### **424** **warning** ***string* is not appropriate for deallocating *string***

This message indicates that a deallocation (`free`, `delete`, or `delete[]`) as specified by the first *string* parameter is inappropriate for the data being freed. [6, Item 5]

The kind of data (specified by the second *string* parameter) is one or more of: `malloc`, `new`, `new[]`, `static`, `auto`, `member`, `modified` or `constant`. These have the meanings as described below:

- `malloc`: data is data obtained from a call to `malloc`, `calloc` or `realloc`.
- `new` and `new[]`: data is data derived from calls to `new`.
- `static`: data is either `static` data within a function or external data.
- `auto`: data is non-static data in a function.
- `member`: data is a component of a structure (and hence can't be independently freed).
- `modified`: data is the result of applying pointer arithmetic to some other pointer. E.g.

```
p = malloc(100);
free( p+1 ); // warning
```

`p+1` is considered modified.

- `constant` data is the result of casting a constant to a pointer. E.g.

```
int *p = (int *) 0x80002;
free(p); // warning
```

See also message [673](#).

Supports AUTOSAR17 Rule A18-5-3

Supports AUTOSAR19 Rule A18-5-3

Supports CERT C MEM34-C - *Only free memory allocated dynamically*

Supports MISRA C 2012 Rule 1.3

Supports MISRA C 2012 Rule 22.2

Supports MISRA C 2004 Rule 1.2

Supports CWE-404 - *Improper Resource Shutdown or Release*

Supports CWE-590 - *Free of Memory not on the Heap*

Supports CWE-762 - *Mismatched Memory Management Routines*

**425** **'message' in processing semantic 'string' at token 'string'**  
**warning** This warning is issued when a syntax error is encountered while processing a semantic option (**-sem**). The 'message' depends upon the error. The first 'string' represents the portion of the semantic being processed. The second 'string' denotes the token being scanned when the error is first noticed.

**426** **call to function *symbol* violates semantic 'string'**  
**warning** This warning message is issued when a user semantic (as defined by **-sem**) is violated. 'string' is the subportion of the semantic that was violated. For example:

```
//lint -sem( f, 1n > 10 && 2n > 10 )
void f( int, int );
...
    f( 2, 20 );
```

results in the message:

```
Call to function 'f(int, int)' violates semantic '(1n>10)'
```

Supports MISRA C 2012 AMD1 Rule 21.13

Supports CWE-119 - *Improper Restriction of Operations within the Bounds of a Memory Buffer*

Supports CWE-176 - *Improper Handling of Unicode Encoding*

Supports CWE-562 - *Return of Stack Variable Address*

Supports CWE-686 - *Function Call With Incorrect Argument Type*

Supports CWE-687 - *Function Call With Incorrectly Specified Argument Value*

Supports CWE-785 - *Use of Path Manipulation Function without Maximum-sized Buffer*

**427** **// comment continued via back-slash**  
**warning** The line that starts a C++ style comment ends with a back-slash causing the next line to be absorbed into the comment, which may not be the intended behavior. If you really intend the next line to be a comment, the line should be started with its own double slash (//) or the entire region replaced with a block comment.

Supports AUTOSAR17 Rule A2-8-1

Supports AUTOSAR19 Rule A2-7-1

Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*

Supports MISRA C 2012 Rule 3.2

**428** **likely indexing before the beginning of an allocation**  
**warning** A negative integer was added to an array or to a pointer to an allocated area (allocated by `malloc`, `operator new`, etc.) This message is not given for pointers whose origin is unknown since a negative subscript is, in general, legal.

The addition could have occurred as part of a subscript operation or as part of a pointer arithmetic operation.

Supports MISRA C 2012 Rule 1.3

**Supports MISRA C 2012 Rule 18.1**

**Supports MISRA C 2004 Rule 1.2**

**Supports CWE-124 - Buffer Underwrite** (*'Buffer Underflow'*)

#### 429 **custodial pointer *symbol* likely not *string* nor returned**

**warning** A pointer of auto storage class was allocated storage, which was neither freed nor returned to the caller. This represents a "memory leak". A pointer is considered custodial if it uniquely points to the storage area. It is not considered custodial if it has been copied. Thus:

```
int *p = new int[20];    // p is a custodial pointer
int *q = p;              // p is no longer custodial
p = new int[20];         // p again becomes custodial
q = p + 0;               // p remains custodial
```

Here `p` does not lose its custodial property by merely participating in an arithmetic operation.

A pointer can lose its custodial property by passing the pointer to a function. If the parameter of the function is typed pointer to `const` or if the function is a library function, that assumption is not made. For example

```
p = malloc(10);
strcpy (p, "hello");
```

Then `p` still has custody of storage allocated.

It is possible to indicate via semantic options that a function will take custody of a pointer. See [9.2.1 custodial\(i\)](#). It is possible to declare that no functions take custody other than those specified in a `-sem` option. See also Flag `ffc` (Functions take custody).

**Supports CERT C ARR00-C - Understand how arrays work**

**Supports CERT C MEM12-C - Consider using a goto chain when leaving a function on error when using and releasing resources**

**Supports CERT C MEM31-C - Free dynamically allocated memory when no longer needed**

**Supports CERT C FIO42-C - Close files when they are no longer needed**

**Supports MISRA C 2012 Rule 22.1**

**Supports CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer**

**Supports CWE-129 - Improper Validation of Array Index**

**Supports CWE-401 - Missing Release of Memory after Effective Lifetime**

**Supports CWE-404 - Improper Resource Shutdown or Release**

**Supports CWE-415 - Double Free**

**Supports CWE-459 - Incomplete Cleanup**

**Supports CWE-771 - Missing Reference to Active Allocated Resource**

**Supports CWE-772 - Missing Release of Resource after Effective Lifetime**

**Supports CWE-773 - Missing Reference to Active File Descriptor or Handle**

**Supports CWE-775 - Missing Release of File Descriptor or Handle after Effective Lifetime**

#### 430 **use of '@' is non-standard**

**warning** Many compilers for embedded systems have a declaration syntax that specifies a location in place of an initial value for a variable. For example:

```
int x @0x2000;
```

specifies that variable `x` is actually location `0x2000`. This message is a reminder that this syntax is non-standard (although quite common). If you are using this syntax on purpose, suppress this message.

#### 432 **suspicious argument to dynamic allocation function**

**warning** The following pattern was detected:

```
malloc( strlen(e+1) )
```

where *e* is some expression. This is suspicious because it closely resembles the commonly used pattern:

```
malloc( strlen(e)+1 )
```

If you really intended to use the first pattern then an equivalent expression that will not raise this error is:

```
malloc( strlen(e)-1 )
```

**Supports CWE-687** - *Function Call With Incorrectly Specified Argument Value*

**433** **allocated area not large enough for pointer (*integer* vs *string*)**  
**warning**

An allocation was assigned to a pointer whose reach extends beyond the area that was allocated. This would usually happen only with library allocation routines such as `malloc` and `calloc`. For example:

```
int *p = malloc(1);
```

This message is also provided for user-declared allocation functions. For example, if a user's own allocation function is provided with the following semantic:

```
-sem(ouralloc,@P==malloc(1n))
```

We would report the same message. Please note that it is necessary to designate that the returned area is freshly allocated (ala `malloc`).

This message is always given in conjunction with the more general Informational Message [826](#).

**Supports CERT C MEM35-C** - *Allocate sufficient memory for an object*

**Supports CWE-131** - *Incorrect Calculation of Buffer Size*

**Supports CWE-190** - *Integer Overflow or Wraparound*

**Supports CWE-467** - *Use of sizeof() on a Pointer Type*

**Supports CWE-680** - *Integer Overflow to Buffer Overflow*

**Supports CWE-789** - *Memory Allocation with Excessive Size Value*

**434** **white space ignored between back-slash and new-line**  
**warning**

According to the C and C++ standards, any back-slash followed immediately by a new-line results in the deletion of both characters. For example:

```
#define A \
34
```

defines `A` to be `34`. If a blank or tab intervenes between the back-slash and the new-line then according to a strict interpretation of the standard you have defined `A` to be a back-slash followed by blank or tab. But this blank is invisible to the naked eye and hence could lead to confusion. Worse, some compilers silently ignore the white-space and the program becomes non-portable.

You should never deliberately place a blank at the end of a line and any such blanks should be removed. If you really need to define a macro to be back-slash blank you can use a comment as in:

```
#define A \ /* commentary */
```

**435** **integral constant '*string*' has precision *integer*, use `+fll` to enable long long**  
**warning**

An integer constant was found that had a precision that was too large for a `long` but would fit within a `long long`. Yet the `+fll` flag that enables the `long long` type was not set.

Check the sizes that you specified for `long` (`-sl#`) and for `long long` (`-sll#`) and make sure they are correct. Turn on `+fll` if your compiler supports `long long`. Otherwise use smaller constants.

**436 preprocessor directive in invocation of macro**

**warning** A function like macro was invoked whose arguments extended for multiple lines, which included preprocessor statements. This is almost certainly an error brought about by a missing right parenthesis.

By the rules of the C and C++ standards, the result of this behavior is undefined. For this reason some compilers treat the apparent preprocessor directive as a directive. However, avoiding this construct is recommended for portability.

**Supports AUTOSAR17 Rule M16-0-5**

**Supports AUTOSAR19 Rule M16-0-5**

**Supports CERT C PRE32-C - Do not use preprocessor directives in invocations of function-like macros**

**Supports MISRA C 2012 Rule 20.6**

**Supports MISRA C++ Rule 16-0-5**

**Supports MISRA C 2004 Rule 19.9**

**437 passing a *class/struct* to an elliptic argument**

**warning** A struct or class is being passed to a function at a parameter position identified by an ellipsis. For example:

```
void g() {
    struct A { int a; } x;
    void f( int, ... );
    f( 1, x );
    ...
}
```

This is sufficiently unusual that it is worth pointing out in the likelihood that this is unintended. The situation becomes more severe in the case of a non-POD struct [10]. In this case the behavior is considered undefined.

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2004 Rule 1.2**

**438 last value assigned to *symbol* not used**

**warning** A value had been assigned to a variable that was not subsequently used. The message is issued either at a return statement or at the end of a block when the variable goes out of scope. For example, consider the following function:

```
void f( int n ) {
    int x = 0, y = 1;
    if( n > 0 ) {
        int z;
        z = x + y;
        if( n > z ) { x = 3; return; }
        z = 12;
    }
}
```

Here we can report that *x* was assigned a value that had not been used by the time the return statement had been encountered. We also report that the most recently assigned value to *z* is unused at the point that *z* goes out of scope. See also Informational message 838 and flags *-fiw* and *-fiz*.

**Supports AUTOSAR17 Rule A0-1-1**

**Supports AUTOSAR17 Rule M0-1-9**

**Supports AUTOSAR19 Rule A0-1-1**

**Supports AUTOSAR19 Rule M0-1-9**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

Supports CERT C MSC13-C - *Detect and remove unused values*

Supports MISRA C 2012 Rule 2.2

Supports MISRA C++ Rule 0-1-6

Supports MISRA C++ Rule 0-1-9

Supports CWE-561 - *Dead Code*

Supports CWE-563 - *Assignment to Variable without Use*

- 440** **for statement condition is inconsistent with modification variable *symbol***  
**warning** A **for** clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd **for** clause expression, is not tested in the 2nd **for** clause expression. For example:

```
for( i = 0; i < 10; j++ )
    ...
```

would draw this complaint since the 'i' of the 2nd expression does not match the 'j' of the third expression.

Supports CERT C MSC21-C - *Use robust loop termination conditions*

Supports MISRA C 2004 Rule 13.5

- 442** **for statement condition and increment directions are inconsistent for variable *symbol***  
**warning** A **for** clause was encountered that appeared to have a parity problem. For example:

```
for( i = 0; i < 10; i-- )
    ...
```

Here the test for i less than 10 seems inconsistent with the 3rd expression of the **for** clause, which decreases the value of i. This same message would be given if i were being increased by the 3rd expression and was being tested for being greater than some value in the 2nd expression.

Supports CERT C MSC21-C - *Use robust loop termination conditions*

- 443** **for statement initializer is inconsistent with modification variable *symbol***  
**warning** A **for** clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd **for** clause expression, does not match the variable that is initialized in the 1st expression. For example:

```
for( ii = 0; i < 10; i++ )
    ...
```

would draw this complaint since the 'ii' of the 1st expression does not match the 'i' of the third expression.

Supports CERT C MSC21-C - *Use robust loop termination conditions*

Supports MISRA C 2004 Rule 13.5

- 444** **for statement condition tests *incremented/decremented* pointer *symbol* for null**  
**warning** The following kind of situation has been detected:

```
for( ... ; p == NULL; p++ )
    ...
```

A loop variable being incremented or decremented would not normally be checked to see if it is NULL. This is more likely a programmer error.

Supports CERT C EXP34-C - *Do not dereference null pointers*

Supports CERT C MSC21-C - *Use robust loop termination conditions*

Supports CWE-476 - *NULL Pointer Dereference*

Supports CWE-690 - *Unchecked Return Value to NULL Pointer Dereference*

**445 reuse of for loop variable *symbol***

**warning** A for loop nested within another for loop employed the same loop variable. For example:

```
for( i = 0; i < 100; i++ ) {
    for( i = 0; i < n; i++ ) { ... }
}
```

Supports CERT C MSC21-C - *Use robust loop termination conditions*

**446 side-effect in initializer list**

**warning** An initializer containing a side effect can be potentially troublesome. For example, the code:

```
void f( int i ) {
    int a[2] = {i++, i++};
}
```

The values of the array elements are unspecified because the order of evaluation is unspecified by the C standard.

Supports MISRA C 2012 Rule 13.1

**447 extraneous whitespace found in include directive for file *file*; Opening file *file***

**warning** A named file was found to contain either leading or trailing whitespace in the `#include` directive. While legal, the ISO Standards allow compilers to define how files are specified or the header is identified, including the appearance of whitespace characters immediately after the `<` or opening `"` or before the `>` or closing `"`. Since filenames tend not to contain leading or trailing whitespace, PC-lint Plus ignores the (apparently) extraneous characters and processes the directive as though the characters were never given. The use of a `-efile` option on either *file* for this message will cause Lint to process `#include`'s with whitespace intact.

**448 possible access *integer* bytes beyond null terminator by '*operator*'**

**warning** Accessing past the terminating nul character is often an indication of a programmer error. For example:

```
char buf[20];
strcpy( buf, "a" );
char c = buf[4]; // legal but suspect
```

Although `buf` has 20 characters, after the `strcpy`, there would be only two that the programmer would normally be interested in.

**449 memory was likely previously deallocated**

**warning** A pointer variable (designated in the message) was freed or deleted in an earlier statement.

Supports CERT C MEM00-C - *Allocate and free memory in the same module, at the same level of abstraction*

Supports CERT C MEM30-C - *Do not access freed memory*

Supports MISRA C 2012 Rule 1.3

Supports MISRA C 2012 Rule 22.2

Supports MISRA C 2004 Rule 1.2

Supports CWE-415 - *Double Free*

Supports CWE-416 - *Use After Free*

Supports CWE-666 - *Operation on Resource in Wrong Phase of Lifetime*

Supports CWE-672 - *Operation on a Resource after Expiration or Release*

Supports CWE-758 - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*

Supports CWE-825 - *Expired Pointer Dereference*



**450 namespace *symbol* declared within an extern "C" region**

**warning** A namespace was declared either with an `extern "C"` specifier or within an `extern "C"` region. The ISO C++ standard leaves the effects of such unspecified. If an `extern "C"` specification is necessary for the declarations within the namespace, it should be inside the namespace rather than outside.

**451 header file '*string*' repeatedly included but has no header guard**

**warning** The file named in the message has already been included in the current module. Moreover, no mechanism to prevent multiple inclusion (such as an include guard or `#pragma once`) was detected. An include guard is one of:

```
#ifndef Name
#define Name Value
...
#endif
```

or

```
#if !defined(X)
#define X Value
...
#endif
```

with nothing but comments before and after this sequence and nothing but comments between the `#if/#ifndef` and the `#define`. `Value` is optional and may contain any expansion text if present. The macro `Name` must not be defined when the header is included for the first time.

This warning may also be accompanied by a [537](#) (repeated include header). Message 537 is often suppressed because if you are working with include guards it is not a helpful message. However, the message [451](#) should be left on in order to check the consistency of the include guards themselves.

**Supports MISRA C 2012 Dir 4.10**

**Supports MISRA C 2004 Rule 19.15**

**452 *type* redefinition with different types *type***

**warning** A `typedef` symbol is being declared to be a different type. This can be legal, especially with multiple modules, but is not good programming practice. It interferes with program legibility.

**453 function *symbol*, previously designated pure, *reason symbol***

**warning** A semantic option designated that the named function, *symbol*, is pure (lacking non-local side-effects): see the `pure` semantic in [Chapter 9 Semantics](#). However, an impurity was detected. Such impurities include calling a function through a function pointer, accessing a volatile variable, modifying a static variable or calling a function whose purity PC-lint Plus cannot verify. *Reason* describes which of these reasons apply and the second *symbol* parameter shows the related variable or function as appropriate.

Despite the inconsistency reported, the function will continue to be regarded as pure.

**454 mutex *symbol* locked without being unlocked**

**warning** A mutex was locked without being subsequently unlocked in the same enclosing scope. For example:

```
#include <mutex>

std::mutex my_mutex;
int my_data = 0;
```

```

void foo() {
    my_mutex.lock();
    ++my_data;
} // 454: my_mutex not unlocked in foo()

```

The location of the lock operation is provided as a supplemental message.

**Supports CERT C CON01-C** - *Acquire and release synchronization primitives in the same module, at the same level of abstraction*

**Supports MISRA C 2012 AMD4 Rule 22.16**

#### 455 *mutex/locker symbol unlocked without being locked*

**warning** A mutex was unlocked without previously being locked in the same enclosing scope. For example:

```

#include <mutex>

std::mutex my_mutex;
int my_data = 0;

void foo() {
    ++my_data;
    my_mutex.unlock(); // 455: my_mutex not locked in foo()
}

```

**Supports CERT C CON01-C** - *Acquire and release synchronization primitives in the same module, at the same level of abstraction*

**Supports MISRA C 2012 AMD4 Rule 22.17**

**Supports CWE-832** - *Unlock of a Resource that is not Locked*

#### 456 *multiple 'string' execution paths are being combined with different lock states*

**warning** The lock state of one or more mutexes could not be determined due to conditional lock state changes whose effect persists after the scope of the condition. This can occur if a mutex is locked without being unlocked in a conditional statement. For example:

```

#include <mutex>

std::mutex my_mutex;
int my_data = 0;

void foo(int i) {
    if (i == 1) { my_mutex.lock(); } // 456
    /* ... */
}

```

Mutex lock and unlock operations should be balanced within the same scope. *String* is one of **break**, **case**, **conditional**, **if**, **iteration**, **switch**, or **try** indicating the type of combined execution path responsible for the inconsistent lock state.

**Supports CERT C CON01-C** - *Acquire and release synchronization primitives in the same module, at the same level of abstraction*

**457** variable '*name*' is '*read/written/atomically read/atomically written*' in function '*name*' of thread '*name*' at *location* which is unprotected with the '*read/write/atomic read/atomic write*' in function '*name*' of thread '*name*' at *location*

A variable was accessed by two threads in an unsafe way outside of the protection of a mutex. An access of the same variable by two threads is considered to be unsafe unless:

- Both accesses are reads, atomic reads, or a combination thereof.
- Both accesses are atomic reads, atomic writes, or a combination thereof.

Any other combination of accesses to the same variable across threads requires acquisition of at least one common mutex by both threads before accessing the variable in either thread. For example, if one thread reads a variable and another thread writes to the same variable, both threads must perform their access while a common mutex is held. Note that it is not sufficient for only one of the pair of unsafe accesses to be protected by a mutex. A variable is assumed to be accessed for writing if it is passed to a function by non-const reference or used to initialize a non-const reference variable even if it is not modified through that reference. This message may be issued for accesses by the same function if the function could execute concurrently.

The message includes the functions, and their corresponding threads, which access the variable, the location and type of access by each function, and the name of the variable that is accessed in an unsafe way. Supplemental messages provide the call path from each thread function and the mutexes held at each point as well as the mutexes held at the point of access.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CERT C CON32-C** - *Prevent data races when accessing bit-fields from multiple threads*

**Supports CERT C CON43-C** - *Do not allow data races in multithreaded code*

**Supports CERT C POS49-C** - *When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed*

**Supports MISRA C 2012 AMD4 Dir 5.1**

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**Supports CWE-366** - *Race Condition within a Thread*

**Supports CWE-667** - *Improper Locking*

**459 warning** function '*name*' whose address was taken has an unprotected '*read/write/atomic read/atomic write*' access of variable '*name*' in function '*name*' at *location*

The specified function accesses a shared variable outside the protection of a locally acquired mutex and had its address taken. Additionally, the program appears to be multi-threaded by virtue of a call to a function with the `thread_create` semantic or the declaration of a function with the `thread` semantic. Supplemental messages provide the location(s) where the address of the function was taken and the call path from the function to the access of the shared variable, if applicable.

If a function's address is taken, it is presumed that the locations from which the function may be called cannot be statically determined. As such, the function needs to have protected access to every shared variable that it accesses.

There are several remedies to such a message. If multiple threads can indeed access this function, then place a mutex lock in the function. If only one thread really accesses this function or if the access is guaranteed to be benign, then, after making sure this condition is commented in the code, the message may be suppressed for the function using the `-estring` option.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**Supports CWE-667** - *Improper Locking*

**460 warning** thread unsafe function '*name*' is called while mutually unprotected in function '*name*' of thread '*name*' at *location* and function '*name*' of thread '*name*' at *location*

The function specified by the first *name* parameter was designated as being **thread\_unsafe** with the **-sem** option. It was being called by the specified functions within the specified threads. The identified calls were not made with the protection of a common mutex. Supplemental messages provide the call path to the function from the specified threads and the mutexes locked at the call points.

Calls to thread unsafe functions need to be protected by exclusive (non-shared) mutex locks if they are to be called by more than one thread.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**Supports CWE-667** - *Improper Locking*

**461** **warning** functions '*name*' and '*name*' of group '*name*' are called mutually unprotected in function '*name*' of thread '*name*' at *location* and function '*name*' of thread '*name*' at *location*

Two functions with the same **thread\_unsafe** group id are being called outside the protection of a common mutex. Functions specified with the same **thread\_unsafe** group id are handled as if they manipulate the same static data and require the protection of exclusive (non-shared) mutex locks when called by multiple threads. Supplemental messages provide the call path to the function from the specified threads and the mutexes locked at the call points.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**Supports CWE-667** - *Improper Locking*

**462** **warning** function '*name*' is called inconsistently with the '*thread\_only/thread\_not*' semantic in function '*name*' of thread '*name*' at *location*

A thread is calling the function identified by the first *name* parameter, either directly or indirectly. The called function has additionally been identified as a function that should not be called by this thread for one of the following reasons:

- The **thread\_not** semantic was applied to the function to indicate that the function should not be called by any thread.
- The **thread\_not** semantic was applied to the function with a list of threads that contains the thread calling the function.
- The **thread\_only** semantic was applied to the function with a list of threads that does not contain the thread calling the function.

The second message parameter specifies the relevant semantic that had been attributed to the function. The calling function and associated thread are provided in the last two parameters. Supplemental messages provide the call graph of the function from the specified thread.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**463** **warning** could not parse '*string*' as a strong type: *string*

This message is issued when a parse failure occurs when parsing a type specified with the **-strong** option. The first *string* contains the type specification that caused the error and the second *string* provides additional information about the error such as "unmatched right parenthesis".

- 464** **buffer arguments '*integer*' and '*integer*' in call to function *symbol* overlap**  
**warning** This is issued when we encounter a function argument expression used in such a way that there will be an attempt to copy its contents onto itself. E.g.

```
sprintf( s, "%s", s );
```

**Supports CWE-687 - Function Call With Incorrectly Specified Argument Value**

- 466** **conversion *to/from* pointer to function with no prototype (*context*)**  
**warning** A pointer to a function without a prototype was assigned to or from another pointer to function. While assigning a pointer to function with a prototype, to one without a prototype is legal in ISO C, unexpected behavior may occur too easily. For example:

```
char *(*pf)();
char *strchr(const char *);
void g() {
    pf = strchr; // Msg 466
    pf(12, 2);   // undefined behavior
}
```

- 473** **argument '*string*' is of insufficient length for array parameter *symbol* declared as *type***  
**warning** This message is issued when a function declared with a constant-sized array parameter is passed an argument which can be determined, using Value Tracking, to either be null or to point to an area that is smaller than the size of the array. For example:

```
void init(unsigned char array[10]);

void *malloc(unsigned);

void foo() {
    unsigned char array1[5];
    unsigned int array2[3];
    unsigned char *pc1 = malloc(8);
    unsigned char *pc2 = (unsigned char *)array2;

    init(array1); // 473 - array1 is 5 bytes, init expects 10
    init(pc1);    // 473 - pc1 points to 8 bytes (or is null)
    init(pc2);    // Okay - assuming ints are 4 bytes or larger
    init(0);      // 473 - null argument
}
```

**Supports CERT C MSC19-C - For functions that return an array, prefer returning an empty array over a null value**

**Supports MISRA C 2012 Rule 17.5**

- 474** **constant switch condition '*string*' not handled by switch**  
**warning** The condition of a `switch` was a constant expression, e.g. `switch(7)`. Furthermore, there is no default case and no case statement that matches the constant expression. '*string*' contains the constant used as the `switch` condition.

**Supports CERT C MSC01-C - Strive for logical completeness**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports CWE-561 - Dead Code**

- 483 switching on a boolean value**  
**warning** At least one standards organization has expressed the perspective, if the expression of a `switch` statement is boolean in nature, `if-else` should be used instead.  
 Supports AUTOSAR17 Rule M6-4-7  
 Supports AUTOSAR19 Rule M6-4-7  
 Supports MISRA C 2012 Rule 16.7  
 Supports MISRA C++ Rule 6-4-7
- 484 stringize operator followed by macro parameter followed by pasting operator**  
**warning** Due to order of evaluation issues, the mixing of stringizing and pasting operators, particularly when appearing in the order `# parameter ##`, results in unspecified behavior.  
 Supports MISRA C 2012 Rule 20.11
- 485 duplicate *partial/complete* initialization of object element**  
**warning** In addition to the behavior being unspecified when the use of designated initializers results in duplicate object initialization, assigning to an array element or structure member more than once in an initializer is typically a logic error.  
 Supports MISRA C 2012 Rule 9.4
- 488 enumerator *symbol* reuses the constant value '*integer*' previously used implicitly by enumerator *symbol***  
**warning** Two enumerators have the same value and at least one received that value implicitly. For example:  

```
enum colors { red, blue, green = 1 };
```

 will elicit this informational message while  

```
enum colors { red, blue = 1, green = 1 };
```

 will not.  
 Supports CERT C INT09-C - *Ensure enumeration constants map to unique values*  
 Supports MISRA C 2012 Rule 8.12
- 489 attempting to modify the contents of a string literal**  
**warning** An assignment to an element of a string literal was seen. Doing so results in undefined behavior.  
 Supports CERT C STR30-C - *Do not attempt to modify string literals*  
 Supports MISRA C 2012 Rule 7.4
- 490 *string***  
**warning** This message is issued as a result of processing a `#warning` preprocessor directive. *string* is the message provided to the directive.
- 491 non-standard use of 'defined' preprocessor operator: *detail***  
**warning** The ISO standards restrict the use of the `defined` preprocessor keyword to either  

```
defined(identifier)
```

```
defined identifier
```

 Additionally, the preprocessor operator may not result from the expansion of another macro. This diagnostic highlights departures from these requirements as non-portable code.  
 Supports AUTOSAR17 Rule M16-1-1

Supports AUTOSAR19 Rule M16-1-1

Supports MISRA C++ Rule 16-1-1

Supports MISRA C 2004 Rule 19.14

#### 492 incomplete format specifier '*string*'

**warning** A format specifier for a `printf/scanf` style function was started but did not contain a conversion specifier. For example:

```
printf("%11", 3LL);
```

will yield the message:

```
incomplete format specifier '%11'
```

Supports CERT C FIO47-C - *Use valid format strings*

Supports CWE-134 - *Use of Externally-Controlled Format String*

Supports CWE-685 - *Function Call With Incorrect Number of Arguments*

Supports CWE-686 - *Function Call With Incorrect Argument Type*

#### 493 position arguments in format strings start counting at 1 (not 0)

**warning** A format specifier for a `printf/scanf` style function attempted to reference the argument at position 0 but positional arguments are indexed at 1 so this is not valid. For example:

```
printf("%0$d", 3);
```

Supports CERT C FIO47-C - *Use valid format strings*

Supports CWE-134 - *Use of Externally-Controlled Format String*

Supports CWE-685 - *Function Call With Incorrect Number of Arguments*

Supports CWE-686 - *Function Call With Incorrect Argument Type*

#### 494 data argument position '*integer*' exceeds the number of data arguments (*integer*)

**warning** A format specifier for a `printf/scanf` style function utilizing positional arguments contained a reference to a non-existent argument, which results in undefined behavior. For example:

```
printf("%2$d", j)
```

will yield the message:

```
data argument position '2' exceeds the number of data arguments (1)
```

Supports CERT C FIO47-C - *Use valid format strings*

Supports CWE-134 - *Use of Externally-Controlled Format String*

Supports CWE-685 - *Function Call With Incorrect Number of Arguments*

Supports CWE-686 - *Function Call With Incorrect Argument Type*

#### 495 format string body contains NUL character

**warning** A format string for a `printf/scanf` style function contains a nul character in the body of the string. The receiving function will not be able to access the portion of the string after this character and its inclusion is likely a mistake. For example:

```
printf("%d\0%d", 1, 2);
```

will elicit this message.

Supports CWE-170 - *Improper Null Termination*

Supports CWE-464 - *Addition of Data Structure Sentinel*

- 496 format string is not null terminated**  
**warning** The format string provided to a `printf`/`scanf` style function is not terminated with a nul character, which will cause the function to read past the end of the string causing undefined behavior.  
**Supports CWE-170** - *Improper Null Termination*  
**Supports CWE-463** - *Deletion of Data Structure Sentinel*
- 497 format string is empty**  
**warning** An empty format string was provided to a `printf` or `scanf` like function. Calling these functions with an empty format string is legal but suspect as there is no effect to doing so.
- 498 unbounded scanf conversion specifier '*string*' may result in buffer overflow**  
**warning** A `%s` or `%[` conversion specifier was encountered in the format string of a `scanf`-like function that did not contain a maximum field width. Since the `%s` and `%[` conversion specifiers read characters into the target buffer until either the maximum field width is reached or a prescribed character is encountered, failing to provide a maximum field width can easily result in buffer overflow. '*string*' contains the unbounded format specifier.  
**Supports CERT C STR31-C** - *Guarantee that storage for strings has sufficient space for character data and the null terminator*  
**Supports CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
**Supports CWE-120** - *Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*  
**Supports CWE-121** - *Stack-based Buffer Overflow*  
**Supports CWE-122** - *Heap-based Buffer Overflow*  
**Supports CWE-123** - *Write-what-where Condition*  
**Supports CWE-125** - *Out-of-bounds Read*  
**Supports CWE-193** - *Off-by-one Error*  
**Supports CWE-676** - *Use of Potentially Dangerous Function*
- 499 using length modifier '*string*' with conversion specifier '*string*' is not supported by ISO C**  
**warning** Within the format for a `printf` or `scanf` like function, a length modifier was combined with a conversion specifier that is not supported by Standard C.  
**Supports CERT C FIO47-C** - *Use valid format strings*  
**Supports CWE-134** - *Use of Externally-Controlled Format String*  
**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*  
**Supports CWE-686** - *Function Call With Incorrect Argument Type*
- 501 negation of value of unsigned type *type* yields a value of unsigned type *type***  
**warning** The unary minus operator was applied to an unsigned type. The resulting value is a positive unsigned quantity and may not be what was intended.  
**Supports CERT C INT02-C** - *Understand integer conversion rules*  
**Supports CWE-192** - *Integer Coercion Error*  
**Supports CWE-197** - *Numeric Truncation Error*
- 502 applying bitwise not to signed quantity**  
**warning** Unary `~` being a bit operator would more logically be applied to unsigned quantities rather than signed quantities.  
**Supports CERT C INT02-C** - *Understand integer conversion rules*  
**Supports CERT C INT16-C** - *Do not make assumptions about representation of signed integers*  
**Supports CWE-192** - *Integer Coercion Error*  
**Supports CWE-197** - *Numeric Truncation Error*



**503 boolean argument to relational**

**warning** Normally a relational would not have a Boolean as argument. An example of this is `a < b < c`, which is technically legal but does not produce the same result as the mathematical expression, which it resembles.

**Supports** CERT C EXP13-C - *Treat relational and equality operators as if they were nonassociative*

**504 unusual shift operation (*string*)**

**warning** Either the quantity being shifted or the amount by which a quantity is to be shifted was derived in an unusual way such as with a bit-wise logical operator, a negation, or with an unparenthesized expression. If the shift value is a compound expression that is not parenthesized, parenthesize it.

**Supports** CWE-1335 - *Incorrect Bitwise Shift of Integer*

**505 redundant left argument to comma**

**warning** The left argument to the comma operator had no side effects in its top-most operator and hence is redundant.

**Supports** CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

**Supports** CERT C MSC13-C - *Detect and remove unused values*

**Supports** MISRA C 2012 Rule 2.2

**Supports** MISRA C 2004 Rule 14.2

**Supports** CWE-561 - *Dead Code*

**506 constant value used in Boolean context (*string*)**

**warning** A value appearing in a Boolean context, such as an operand to `&&`, `||`, or `!` or a condition of `?:`, `if`, `do`, `for`, or `while`, was found to have a constant value and hence will evaluate the same way each time. This message will not be issued in contexts where a constant is required or expected, such as a `static_assert` condition or an array size.

Some special cases are reported using other messages:

- Message 716: `while (1)` or `while (true)`
- Message 717: `do ... while (0)` or `do ... while (false)`
- Message 2441: result of address-of operator used in Boolean context
- Message 3416: `this` pointer used in Boolean context

**Supports** CWE-561 - *Dead Code*

**507 explicit cast from *type* to *type* (*integer* bits to *integer* bits)**

**warning** A cast was made to an integral quantity from a pointer and according to other information given or implied it would not fit. For example, a cast to an `unsigned int` was specified and information provided by the options indicate that a pointer is larger than an `int`. Two *integers* are supplied. The first is the size in bytes of the pointer and the second is the size in bytes of the integer.

**511 explicit cast from *type* to *type* (*integer* bits to *integer* bits)**

**warning** A cast was made from an integral type to a pointer and the size of the quantity was too large to fit into the pointer. For example if a `long` is cast to a pointer and if options indicate that a `long` is larger than a pointer, this warning would be reported.

**513 the option '*string*' is not currently supported**

**warning** The specified option is not supported in this version of PC-lint Plus but may be available in a future version.

**514 boolean argument to *arithmetic/bitwise* operator '*operator*'**

**warning** An argument to an arithmetic operator (+ - / \* %) or a bit-wise logical operator (| & ^) was a Boolean. This can often happen by accident as in:

```
if( flags & 4 == 0 )
```

where the ==, having higher precedence than &, is done first (to the puzzlement of the programmer).

**Supports CERT C EXP46-C** - Do not use a bitwise operator with a Boolean-like operand

**Supports CWE-480** - Use of Incorrect Operator

**517 defined not K&R**

**warning** The **defined** preprocessor directive (which was not present in K&R C) was employed and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use **defined**.

**518 expected parenthesis around type name in *context* expression**

**warning** **sizeof** *type* is not strict C. **sizeof**(*type*) or **sizeof** *expression* are both permissible.

**519 explicit cast from *type* to *type* (*integer* bits to *integer* bits)**

**warning** An attempt was made to cast a pointer to a pointer of unequal size. This could occur for example in a P model where pointers to functions require 4 bytes whereas pointers to data require only 2. This error message can be circumvented by first casting the pointer to an integral quantity (**int** or **long**) before casting to a pointer.

**520 first clause of for statement lacks side effects**

**warning** The first expression of a **for** clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning [522](#).

**Supports MISRA C 2012 Rule 2.2**

**521 third clause of for statement lacks side effects**

**warning** The third expression of a **for** clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning [522](#).

**Supports MISRA C 2012 Rule 2.2**

**522 highest operation, *string* '*name*', lacks side effects**

**warning** If a statement consists only of an expression, it should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). For example, if operator \* is the built-in operator, the statement **\*p++**; draws this message with *string* equal to operator and *name* equal to \*. But note that **p++**; does not. This is because the highest operator in the former case is '\*', which has no side effects whereas **p++** does. It is possible for a function to have no side-effects. Such a function is called pure. See the discussion of the pure semantic in Section [9.2.1 Possible Semantics](#). For example:

```
void f() { int n = 3; n++; }
void g() { f(); }
```

will trigger this message with *string* in the message equal to function and *name* equal to *f*.

The definition of pure and impure functions and function calls that have side effects are given in the discussion of the pure semantic in Chapter 9 [Semantics](#).

**Supports MISRA C 2004 Rule 14.2 (Req)**

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports MISRA C 2012 Rule 2.2**

**Supports MISRA C 2004 Rule 14.2**

**Supports CWE-480** - *Use of Incorrect Operator*

**Supports CWE-482** - *Comparing instead of Assigning*

**Supports CWE-561** - *Dead Code*

### 523 expression statement involving *string* '*name*' lacks side effects

**warning** This message is similar to 522 but is issued only if the entire statement lacks side effects. For example:

```
void foo() {
    int i = 0;
    i++ + 1;
}
```

While the operator `+` lacks side effects, the statement doesn't so 522 will be issued here but not 523.

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports CWE-561** - *Dead Code*

### 524 implicit truncation from *type* to *type*

**warning** There is a possible loss of a fraction in converting from a float to an integral quantity. Use of a cast will suppress this message.

### 525 unexpected negative indentation

**warning** The current line was found to be negatively indented (i.e., not indented as much) from the indicated line. The latter corresponds to a clause introducing a control structure, and statements and other control clauses and braces within its scope are expected to have no less indentation. If tabs within your program are other than 8 blanks you should use the `-t` option (See Section 17.4 [Indentation Checking](#)).

**Supports CWE-483** - *Incorrect Block Delimitation*

### 526 symbol *symbol* is not defined

**warning** The named external was referenced but not defined. This message is not issued for library symbols and is suppressed for unit checkout (`-unit_check` option). Please note that a declaration, even one bearing prototype information is not a definition. See the glossary at the beginning of this chapter. If the *symbol* is a library symbol, make sure that it is declared in a header file that you're including. Also make sure that the header file is regarded by PC-lint Plus as a Library Header file. Alternatively, the symbol may be declared in a Library Module. See Section 5.1 [Library Header Files](#) and Section 5.2 [Library Modules](#) for a further discussion.

### 527 statement is unreachable due to unconditional transfer of control by '*string*' statement

**warning** A portion of the program cannot be reached. The control mechanism responsible for unconditionally diverting flow away from the specified area is given by *string*.

**Supports AUTOSAR17 Rule M0-1-1**

**Supports AUTOSAR19 Rule M0-1-1**

**Supports CERT C DCL41-C** - *Do not declare variables inside a switch statement before the first case label*

Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

Supports MISRA C 2012 Rule 2.1

Supports MISRA C++ Rule 0-1-1

Supports MISRA C 2004 Rule 14.1

Supports CWE-561 - *Dead Code*

**528** static symbol *symbol* not referenced  
 warning The named static variable or static function was not referenced in the module after having been declared.  
 Supports AUTOSAR17 Rule M0-1-3  
 Supports AUTOSAR17 Rule M0-1-4  
 Supports AUTOSAR17 Rule M0-1-10  
 Supports AUTOSAR17 Rule A0-1-3  
 Supports AUTOSAR19 Rule M0-1-3  
 Supports AUTOSAR19 Rule M0-1-4  
 Supports AUTOSAR19 Rule M0-1-10  
 Supports AUTOSAR19 Rule A0-1-3  
 Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*  
 Supports MISRA C 2012 AMD4 Rule 2.8  
 Supports MISRA C++ Rule 0-1-3  
 Supports MISRA C++ Rule 0-1-4  
 Supports MISRA C++ Rule 0-1-10  
 Supports CWE-561 - *Dead Code*

**529** local variable *symbol* declared in *symbol* not subsequently referenced  
 warning The named variable was declared but not referenced in a function.  
 Supports AUTOSAR17 Rule M0-1-3  
 Supports AUTOSAR17 Rule M0-1-4  
 Supports AUTOSAR19 Rule M0-1-3  
 Supports AUTOSAR19 Rule M0-1-4  
 Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*  
 Supports CERT C MSC13-C - *Detect and remove unused values*  
 Supports MISRA C 2012 AMD4 Rule 2.8  
 Supports MISRA C++ Rule 0-1-3  
 Supports MISRA C++ Rule 0-1-4  
 Supports CWE-561 - *Dead Code*

**530** likely using an uninitialized value  
 warning An auto variable was used before it was initialized.  
 Supports AUTOSAR17 Rule M8-5-1  
 Supports AUTOSAR19 Rule A8-5-0  
 Supports CERT C EXP33-C - *Do not read uninitialized memory*  
 Supports MISRA C 2012 Rule 9.1  
 Supports MISRA C++ Rule 8-5-1  
 Supports MISRA C 2004 Rule 9.1  
 Supports CWE-457 - *Use of Uninitialized Variable*  
 Supports CWE-758 - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*  
 Supports CWE-908 - *Use of Uninitialized Resource*

**531** width of bit-field *symbol* (*integer* bits) exceeds size of its type (*integer* bit(s))  
 warning The size given for a bit field of a structure exceeds the size of an `int`.

- 533 function *symbol* should return a value**  
**warning** A function declared as returning non-void either contains a **return** statement that is missing an expression or control may reach the end of the function without a value being returned.  
**Supports AUTOSAR17 Rule A8-4-2**  
**Supports AUTOSAR19 Rule A8-4-2**  
**Supports CERT C MSC37-C - *Ensure that control never reaches the end of a non-void function***  
**Supports MISRA C 2012 Rule 17.4**  
**Supports MISRA C++ Rule 8-4-3**  
**Supports MISRA C 2004 Rule 16.8**  
**Supports CWE-758 - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior***
- 534 ignoring return value of function *symbol***  
**warning** A function that returns a value is called just for side effects as, for example, in a statement by itself or the left-hand side of a comma operator. Try: `(void) function();` to call a function and ignore its return value.  
**Supports AUTOSAR17 Rule A0-1-2**  
**Supports AUTOSAR19 Rule A0-1-2**  
**Supports CERT C EXP12-C - *Do not ignore values returned by functions***  
**Supports CERT C ERR33-C - *Detect and handle standard library errors***  
**Supports CERT C POS54-C - *Detect and handle POSIX library errors***  
**Supports MISRA C 2012 Dir 4.7**  
**Supports MISRA C 2012 Rule 17.7**  
**Supports MISRA C++ Rule 0-1-7**  
**Supports MISRA C++ Rule 0-3-2**  
**Supports MISRA C 2004 Rule 16.10**  
**Supports CWE-252 - *Unchecked Return Value***  
**Supports CWE-253 - *Incorrect Check of Function Return Value***  
**Supports CWE-391 - *Unchecked Error Condition***
- 537 repeated include file '*file*'**  
**warning** The file whose inclusion within a module is being requested has already been included in this compilation. The file is processed normally even if the message is given. If it is your standard practice to repeat included files then simply suppress this message.
- 539 unexpected positive indentation**  
**warning** The current line was found to be positively indented from a clause that did not control the line in question. For example:
- ```

    if( n > 0 )
        x = 3;
        y = 4;

```
- will result in this warning being issued for `y = 4; .`  
**Supports CWE-483 - *Incorrect Block Delimitation***
- 540 initializer-string for char array is too long, array size is *integer* but initializer has size *integer***  
**warning** (including the null terminating character)  
A string initializer required more space than what was allocated.

**542 excessive size for bit field**

**warning** An attempt was made to assign a value into a bit field that appears to be too small.

You may get this message unexpectedly if the base of the bit field is an `int`. For example:

```
struct { int b : 1 } s;
s.b = 1; /* Warning -- requires 0 or -1 */
```

The solution in this case is to use `'unsigned'` rather than `'int'` in the declaration of `b`.

**544 *string* directive not followed by EOL**

**warning** The preprocessor directive `#endif` should be followed by an end-of-line. Some compilers specifically allow commentary to follow the `#endif`. If you are following that convention simply turn this error message off.

**Supports AUTOSAR17 Rule M16-0-8**

**Supports AUTOSAR19 Rule M16-0-8**

**Supports MISRA C 2012 Rule 20.3**

**Supports MISRA C 2012 Rule 20.13**

**Supports MISRA C++ Rule 16-0-8**

**Supports MISRA C 2004 Rule 19.16**

**545 taking address of array**

**warning** An attempt was made to take the address of an array name. At one time such an expression was officially illegal (K&R C [7]), was not consistently implemented, and was, therefore, suspect. However, the expression is legal in ANSI/ ISO C and designates a pointer to an array. For example, given

```
int a[10];
int (*p) [10];
```

Then `a` and `&a`, as pointers, both represent the same bit pattern, but whereas `a` is a pointer to `int`, `&a` is a pointer to an array of 10 integers. Of the two only `&a` may be assigned to `p` without complaint. If you are using the `&` operator in this way, we recommend that you disable this message.

**546 explicitly taking address of function**

**warning** An attempt was made to take the address of a function name. Since names of functions by themselves are promoted to address, the use of the `&` is redundant and could be erroneous.

**547 redefinition of macro *name* conflicts with previous definition**

**warning** The indicated symbol had previously been defined (via `#define`) to some other value.

**Supports MISRA C 2012 Rule 5.4**

**548 if statement has no body or else**

**warning** A construct of the form `if(e);` was found, and it was not followed by an `else`. This is almost certainly an unwanted semi-colon as it inhibits the `if` from having any effect.

**549 explicit cast from *type* to *type***

**warning** A cast was made from a pointer to some enumerated type or from an enumerated type to a pointer. This is probably an error. Check your code and if this is not an error, then cast the item to an intermediate form (such as an `int` or a `long`) before making the final cast.

- 550 local variable *symbol* declared in *symbol* not subsequently accessed**  
**warning** A variable (local to some function) was not accessed. This means that the value of a variable was never used. Perhaps the variable was assigned a value but was never used. Note that a variable's value is not considered accessed by autoincrementing or autodecrementing unless the autoincrement/decrement appears within a larger expression, which uses the resulting value. The same applies to a construct of the form: *var* += *expression*. If an address of a variable is taken, its value is assumed to be accessed. However, casting that address to a non-pointer causes Lint to forget this sense of "accessed-ness." An array, **struct** or **union** is considered accessed if any portion thereof is accessed.  
**Supports AUTOSAR17 Rule M0-1-4**  
**Supports AUTOSAR19 Rule M0-1-4**  
**Supports MISRA C++ Rule 0-1-4**
- 551 static variable *symbol* not accessed**  
**warning** A variable (declared **static** at the module level) was not accessed though the variable was referenced. See the explanation under message [550](#) (above) for a description of "access".  
**Supports AUTOSAR17 Rule M0-1-4**  
**Supports AUTOSAR19 Rule M0-1-4**  
**Supports MISRA C++ Rule 0-1-4**
- 552 external variable *symbol* not accessed**  
**warning** An external variable was not accessed though the variable was referenced. See the explanation under message [550](#) above for a description of "access". This message is not issued for library symbols and is suppressed for unit checkout (**-unit\_check** option).  
**Supports AUTOSAR17 Rule M0-1-4**  
**Supports AUTOSAR19 Rule M0-1-4**  
**Supports MISRA C++ Rule 0-1-4**
- 553 undefined preprocessor variable *name*, assumed 0**  
**warning** The indicated variable had not previously been defined within a **#define** statement and yet it is being used in a preprocessor condition of the form **#if** or **#elif**. Conventionally all variables in preprocessor expressions should be pre-defined. The value of the variable is assumed to be 0.  
**Supports AUTOSAR17 Rule M16-0-7**  
**Supports AUTOSAR19 Rule M16-0-7**  
**Supports MISRA C 2012 Rule 20.9**  
**Supports MISRA C++ Rule 16-0-7**  
**Supports MISRA C 2004 Rule 19.11**
- 555 #elif not K&R**  
**warning** The **#elif** directive was used and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use **#elif**.
- 557 invalid conversion specifier '*string*'**  
**warning** The format string supplied to a **printf**/**scanf** style function was not recognized. It is neither a standard format nor a recognized common extension (see message [816](#)).  
**Supports CERT C FIO47-C - Use valid format strings**  
**Supports MISRA C 2012 Rule 1.3**  
**Supports MISRA C 2004 Rule 1.2**  
**Supports CWE-134 - Use of Externally-Controlled Format String**  
**Supports CWE-685 - Function Call With Incorrect Number of Arguments**

Supports **CWE-686** - *Function Call With Incorrect Argument Type*

- 558** **too few data arguments for format string (*integer* missing)**  
**warning** The number of arguments supplied to a `printf/scanf` style function was less than the number expected. The number of missing arguments is given by *integer*. See also message [719](#).  
 Supports **CERT C DCL10-C** - *Maintain the contract between the writer and caller of variadic functions*  
 Supports **CERT C FIO47-C** - *Use valid format strings*  
 Supports **MISRA C 2012 Rule 1.3**  
 Supports **MISRA C 2004 Rule 1.2**  
 Supports **CWE-134** - *Use of Externally-Controlled Format String*  
 Supports **CWE-628** - *Function Call with Incorrectly Specified Arguments*  
 Supports **CWE-685** - *Function Call With Incorrect Number of Arguments*  
 Supports **CWE-686** - *Function Call With Incorrect Argument Type*

- 559** **format '*string*' specifies type *type* which is inconsistent with argument no. *integer* of *string type***  
**warning** The argument corresponding to a conversion specifier in a `printf/scanf` style function was not of the correct type. The format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
extern char * buffer;
sprintf(buffer, "%f", 371);
```

will result in the message:

```
format '\%f' specifies type 'double' which is inconsistent with
argument no. 3 of type 'int'
```

For conflicting integer types that differ only in signedness (e.g. `int` vs. `unsigned int`) or exact type (e.g. `int` vs. `long` when both are the same size) [705](#) is given instead. For conflicting pointer to integer types that differ only in the signedness or exact type of the pointee, [706](#) is given.

Supports **CERT C DCL11-C** - *Understand the type issues associated with variadic functions*  
 Supports **CERT C INT00-C** - *Understand the data model used by your implementation(s)*  
 Supports **CERT C FIO47-C** - *Use valid format strings*  
 Supports **CWE-134** - *Use of Externally-Controlled Format String*  
 Supports **CWE-685** - *Function Call With Incorrect Number of Arguments*  
 Supports **CWE-686** - *Function Call With Incorrect Argument Type*  
 Supports **CWE-843** - *Access of Resource Using Incompatible Type ('Type Confusion')*

- 563** **label *name* not referenced**  
**warning** *name* appeared as a label but there was no statement that referenced this label.  
 Supports **CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*  
 Supports **MISRA C 2012 Rule 2.6**  
 Supports **CWE-561** - *Dead Code*

- 564** **variable *symbol* depends on order of evaluation**  
**warning** The named variable was both modified and accessed in the same expression in such a way that the result depends on whether the order of evaluation is left-to-right or right-to-left. One such example is: `n + n++` since there is no guarantee that the first access to `n` occurs before the increment of `n`. This message is also triggered by the potential modification of an object by a call to a function whose corresponding parameter is a non-`const` reference or a non-`const` pointer. Volatile variables are also checked for repeated use in an expression.  
 Supports **AUTOSAR17 Rule A5-0-1**



**Supports AUTOSAR19 Rule A5-0-1**

**Supports CERT C EXP10-C** - *Do not depend on the order of evaluation of subexpressions or the order in which side effects take place*

**Supports CERT C EXP30-C** - *Do not depend on the order of evaluation for side effects*

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2012 Rule 13.2**

**Supports MISRA C++ Rule 5-0-1**

**Supports MISRA C 2004 Rule 1.2**

**Supports MISRA C 2004 Rule 12.2**

**Supports CWE-758** - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*

#### **565 declaration of tag *type* will not be visible outside of this function**

**warning** The named tag appeared in a function prototype in a C module and does not correspond to a previously seen declaration in an outer (file-level) scope. Move the tag declaration before the function if the intention is for it to be visible outside the function.

#### **566 inconsistent or redundant *length modifier/flag 'string'* used with '*string*' conversion specifier**

**warning** This message is given for format specifiers within formats for the `printf/scanf` family of functions. The indicated *length modifier* or *flag character* found in a format specifier either has no effect or is not allowed to be combined with the provided conversion specifier. For example:

```
printf("%+u", 123);
```

will yield the message:

**inconsistent or redundant flag '+' used with 'u' conversion specifier**

because the + flag is valid only with signed conversions, its use with other conversions results in undefined behavior.

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

#### **567 expected minimum field width for flag '*string*'**

**warning** This message is given for format specifiers within formats for the `printf/scanf` family of functions. A numeric field or asterisk was expected at a particular point in the scanning of the format. For example: `%-d` requests left justification of a decimal integer within a format field. But since no field width is given, the request is meaningless.

#### **568 nonnegative quantity is never less than zero**

**warning** Comparisons of the form:

```
u >= 0      0 <= u
u < 0       0 >  u
```

are suspicious if `u` is an unsigned quantity or a quantity judged to be never less than 0. See also message [775](#).

**Supports CWE-398** - *Indicator of Poor Code Quality*

#### **569 loss of information (*context*) in implicit conversion from *type integer (integer bits)* to *type (integer bits)***

**warning** An assignment (or implied assignment, see *context*) was made from a constant to an integral variable that is not large enough to hold the constant. Examples include placing a hex constant whose bit requirement is

such as to require an `unsigned int` into a variable typed as `int`. The number of bits given does not count the sign bit.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

**570 negative *type integer* loses sign during implicit conversion (*context*) to *type***

**warning** An assignment (or implied assignment, see *context*) is being made from a negative constant into an unsigned quantity. Casting the constant to `unsigned` will remove the diagnostic but is this what you want? If you are assigning all ones to an `unsigned`, remember that `~0` represents all ones and is more portable than `-1`.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

**571 cast from *type* to *type* results in sign extension**

**warning** Usually this warning is issued for casts of the form:

```
(unsigned) ch
```

where `ch` is declared as `char` and `char` is signed. Although the cast may appear to prevent sign extension of `ch`, it does not. Following the normal conversion rules of C, if `ch` is negative then it cannot be represented in an `unsigned` type and so a quantity of  $2^{**n}$  is added to the signed quantity where  $n$  is the number of bits in the destination. If  $2^{**m}$  were added, where  $m$  is the number of bits in the source, i.e. `ch`, then the sign extension would not occur. To suppress sign extension you may use:

```
(unsigned char) ch
```

Otherwise, if sign extension is what you want and you just want to suppress the warning in this instance you may use:

```
(unsigned) (int) ch
```

Although these examples have been given in terms of casting a `char`, this message will also be given whenever this cast is made upon a signed quantity whose size is less than the casted type. Examples include signed bit fields, expressions involving `char`, and expressions involving `short` when this type is smaller than `int` or a direct cast of an `int` to an `unsigned long` (if `int` is smaller than long). This message is not issued for constants or for expressions involving bit operations.

**Supports CERT C STR34-C** - *Cast characters to unsigned char before converting to larger integer sizes*

**Supports CWE-704** - *Incorrect Type Conversion or Cast*

**572 excessive shift value (precision *integer* shifted right by *integer*)**

**warning** A quantity is being shifted to the right whose precision is equal to or smaller than the shifted value. For example,

```
ch >> 10
```

will elicit this message if `ch` is typed `char` and where `char` is less than 10 bits wide (the usual case). To suppress the message in this case you may cast the shifted quantity to a type whose length is at least the length of the shift value.

The precision of a constant (including enumeration constants) is determined from the number of bits required in its binary representation. The precision does not change with a cast so that `(unsigned) 1 >> 3` still yields the message. But normally the only way an expression such as `1 >> 3` can legitimately occur is via a macro. In this case use `-emacro`.

**Supports CWE-1335** - *Incorrect Bitwise Shift of Integer*

**573 signed-unsigned mix with divide**

**warning** one of the operands to `/` or `%` was signed and the other unsigned; moreover the signed quantity could be negative. For example:

```
u / n
```

where `u` is unsigned and `n` is signed will elicit this message whereas:

```
u / 4
```

will not, even though `4` is nominally an `int`. It is not a good idea to mix unsigned quantities with signed quantities in any case (a [737](#) will also be issued) but, with division, a negative value can create havoc. For example, the innocent looking:

```
n = n / u
```

will, if `n` is `-2` and `u` is `2`, not assign `-1` to `n` but will assign some very large value.

To resolve this problem, either cast the integer to **unsigned** if you know it can never be less than zero or cast the **unsigned** to an integer if you know it can never exceed the maximum integer.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

**574 signed-unsigned mix with relational**

**warning** The four relational operators are:

```
> >= < <=
```

One of the operands to a relational operator was signed and the other unsigned; also, the signed quantity could be negative. For example:

```
if( u > n ) ...
```

where `u` is unsigned and `n` is signed will elicit this message whereas:

```
if( u > 12 ) ...
```

will not (even though `12` is officially an `int` it is obvious that it is not negative). It is not a good idea to mix unsigned quantities with signed quantities in any case (a [737](#) will also be issued) but, with the four relationals, a negative value can produce obscure results. For example, if the conditional:

```
if( n < 0 ) ...
```

is true then the similar appearing:

```
u = 0;
if( n < u ) ...
```

is false because the promotion to unsigned makes `n` very large.

To resolve this problem, either cast the integer to **unsigned** if you know it can never be less than zero or cast the **unsigned** to an `int` if you know it can never exceed the maximum `int`.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

- 575 enumeration constant exceeds range for integers**  
**warning** For many compilers the value of an enumeration constant is limited to those values that can fit within a signed or unsigned int.
- 576 excess elements in *string* initializer**  
**warning** In a brace-enclosed initializer, there are more items than there are elements of the aggregate, which will result in undefined behavior as this is a constraint violation in C. For example:
- ```
int array[3] = {1, 2, 3, 4};
```
- In C++, an error is emitted instead.  
**Supports CERT C ARR02-C** - *Explicitly specify array bounds, even if implicitly defined by an initializer*  
**Supports MISRA C 2004 Rule 9.2**  
**Supports CWE-665** - *Improper Initialization*
- 578 declaration of *symbol* hides *string***  
**warning** A local symbol has the identical name as a variable or field specified by *detail*. This could be dangerous. Was this deliberate? It is usually best to rename the local symbol.  
**Supports CERT C DCL01-C** - *Do not reuse variable names in subscope*  
**Supports MISRA C 2012 Rule 5.3**  
**Supports MISRA C++ Rule 2-10-2**  
**Supports MISRA C 2004 Rule 5.2**
- 579 parameter preceding ellipsis cannot be '*string*'**  
**warning** When an ellipsis is used, the type of the parameter immediately preceding the ellipsis should not be a type that would undergo a default promotion such as **char**, **short** or **float**, a reference type, or a parameter declared with the **register** keyword. The C and C++ standards explicitly state that attempting to extract variable arguments from a call to such a function results in undefined behavior. The *string* parameter is one of: a promotable integer type, a promotable floating point type, a reference type, or declared as **register**.
- 580 redeclaration of function *symbol* causes loss of prototype**  
**warning** A declaration of a function within a block hides a declaration in an outer scope in such a way that the inner declaration has no prototype and the outer declaration does. A common misconception is that the resulting declaration is a composite of both declarations but this is only the case when the declarations are in the same scope not within nested scopes. If you do not care about prototypes you may suppress this message. You will still receive other type-difference warnings.
- 583 comparing type '*type*' with EOF**  
**warning** This message is issued when some form of character is compared against the EOF macro. EOF is normally defined to be -1. For example:
- ```
while( (ch = getchar()) != EOF ) ...
```
- If **ch** is defined to be an **int** all is well. If however it is defined to be some form of **char**, then trouble might ensue. If **ch** is an **unsigned char** then it can never equal EOF. If **ch** is a **signed char** then you could get a premature termination because some data character happened to be all ones.
- Note that **getchar** returns an **int**. The reason it returns an **int** and not a **char** is because it must be capable of returning 257 different values (256 different characters plus EOF, assuming an 8-bit character). Once this value is assigned to a **char** only 256 values are then possible – a clear loss of information.

**584 trigraph sequence (??*character*) detected**

**warning** This message is issued whenever a trigraph sequence is detected and the trigraph processing has been turned off (with a `-ftg`). If this is within a string (or character) constant then the trigraph nature of the sequence is ignored. That is, three characters are produced rather than just one. This is useful if your compiler does not process trigraph sequences and you want linting to mirror compilation. Outside of a string we issue this warning but we do translate the sequence since it cannot make syntactic sense in its raw state.

**Supports AUTOSAR17 Rule A2-5-1**

**Supports AUTOSAR19 Rule A2-5-1**

**Supports CERT C PRE07-C - Avoid using repeated question marks**

**Supports MISRA C 2012 Rule 4.2**

**Supports MISRA C++ Rule 2-3-1**

**Supports MISRA C 2004 Rule 4.2**

**585 the sequence (??*character*) is not a valid trigraph sequence**

**warning** This warning is issued whenever a pair of '?' characters is seen within a string (or character) constant but that pair is not followed by a character, which would make the triple a valid Trigraph sequence. Did the programmer intend this to be a Trigraph sequence and merely err? Even if no Trigraph were intended it can easily be mistaken by the reader of the code to be a Trigraph. Moreover, what assurances do we have that in the future the invalid Trigraph might not become a valid Trigraph and change the meaning of the string? To protect yourself from such an event you may place a backslash between the '?' characters. Alternatively you may use concatenation of string constants. For example:

```
pattern = "(???) ???-????";           // warning 585
pattern = "(?\\?) ?\\?-?\\?\\?";      // no warning
#define Q "?"
pattern="(" Q Q Q ") " Q Q Q "- " Q Q Q Q; //no warning
```

**586 *string 'name'* is deprecated. *string***

**warning** The *name* has been deprecated by some use of the deprecate option. See `-deprecate`. The first *string* is one of the allowed categories of deprecation. The trailing *string* is part of the deprecate option and should explain why the facility has been deprecated.

**Supports AUTOSAR17 Rule A0-4-2**

**Supports AUTOSAR17 Rule A1-1-1**

**Supports AUTOSAR17 Rule A2-14-3**

**Supports AUTOSAR17 Rule A3-9-1**

**Supports AUTOSAR17 Rule A5-2-1**

**Supports AUTOSAR17 Rule A5-2-4**

**Supports AUTOSAR17 Rule A7-1-4**

**Supports AUTOSAR17 Rule A7-4-1**

**Supports AUTOSAR17 Rule A15-5-2**

**Supports AUTOSAR17 Rule A16-0-1**

**Supports AUTOSAR17 Rule A16-6-1**

**Supports AUTOSAR17 Rule A16-7-1**

**Supports AUTOSAR17 Rule M17-0-5**

**Supports AUTOSAR17 Rule A18-0-2**

**Supports AUTOSAR17 Rule M18-0-3**

**Supports AUTOSAR17 Rule M18-0-4**

**Supports AUTOSAR17 Rule M18-0-5**

**Supports AUTOSAR17 Rule A18-0-3**

**Supports AUTOSAR17 Rule M18-2-1**

**Supports AUTOSAR17 Rule A18-5-1**

Supports AUTOSAR17 Rule M18-7-1  
 Supports AUTOSAR17 Rule M19-3-1  
 Supports AUTOSAR17 Rule M27-0-1  
 Supports AUTOSAR19 Rule A0-4-2  
 Supports AUTOSAR19 Rule A1-1-1  
 Supports AUTOSAR19 Rule A2-11-1  
 Supports AUTOSAR19 Rule A2-13-3  
 Supports AUTOSAR19 Rule A3-9-1  
 Supports AUTOSAR19 Rule A5-2-1  
 Supports AUTOSAR19 Rule A5-2-4  
 Supports AUTOSAR19 Rule A7-1-4  
 Supports AUTOSAR19 Rule A7-4-1  
 Supports AUTOSAR19 Rule A15-5-2  
 Supports AUTOSAR19 Rule A16-0-1  
 Supports AUTOSAR19 Rule A16-6-1  
 Supports AUTOSAR19 Rule A16-7-1  
 Supports AUTOSAR19 Rule M17-0-5  
 Supports AUTOSAR19 Rule A18-0-2  
 Supports AUTOSAR19 Rule M18-0-3  
 Supports AUTOSAR19 Rule M18-0-4  
 Supports AUTOSAR19 Rule M18-0-5  
 Supports AUTOSAR19 Rule A18-0-3  
 Supports AUTOSAR19 Rule M18-2-1  
 Supports AUTOSAR19 Rule A18-5-1  
 Supports AUTOSAR19 Rule M18-7-1  
 Supports AUTOSAR19 Rule M19-3-1  
 Supports AUTOSAR19 Rule A26-5-1  
 Supports AUTOSAR19 Rule M27-0-1  
 Supports CERT C EXP43-C - *Avoid undefined behavior when using restrict-qualified pointers*  
 Supports CERT C INT05-C - *Do not use input functions to convert character data if they cannot handle all possible inputs*  
 Supports CERT C STR07-C - *Use the bounds-checking interfaces for string manipulation*  
 Supports CERT C ENV33-C - *Do not call system()*  
 Supports CERT C SIG00-C - *Mask signals handled by noninterruptible signal handlers*  
 Supports CERT C SIG01-C - *Understand implementation-specific details regarding signal handler persistence*  
 Supports CERT C SIG02-C - *Avoid using signals to implement normal functionality*  
 Supports CERT C ERR04-C - *Choose an appropriate termination strategy*  
 Supports CERT C ERR06-C - *Understand the termination behavior of assert() and abort()*  
 Supports CERT C ERR07-C - *Prefer functions that support error checking over equivalent functions that don't*  
 Supports CERT C ERR34-C - *Detect errors when converting a string to a number*  
 Supports CERT C MSC06-C - *Beware of compiler optimizations*  
 Supports CERT C MSC18-C - *Be careful while handling sensitive data, such as passwords, in program code*  
 Supports CERT C MSC24-C - *Do not use deprecated or obsolescent functions*  
 Supports CERT C MSC30-C - *Do not use the rand() function for generating pseudorandom numbers*  
 Supports CERT C MSC33-C - *Do not pass invalid data to the asctime() function*  
 Supports CERT C WIN00-C - *Be specific when dynamically loading libraries*  
 Supports CERT C WIN01-C - *Do not forcibly terminate execution*  
 Supports CERT C WIN02-C - *Restrict privileges when spawning child processes*  
 Supports CERT C CON33-C - *Avoid race conditions when using library functions*  
 Supports CERT C CON37-C - *Do not call signal() in a multithreaded program*  
 Supports CERT C POS04-C - *Avoid using PTHREAD\_MUTEX\_NORMAL type mutex locks*  
 Supports CERT C POS33-C - *Do not use vfork()*  
 Supports CERT C POS44-C - *Do not use signals to terminate threads*  
 Supports CERT C POS47-C - *Do not use threads that can be canceled asynchronously*  
 Supports MISRA C 2012 AMD2 Rule 1.4

Supports MISRA C 2012 AMD2 Rule 21.3  
 Supports MISRA C 2012 AMD2 Rule 21.8  
 Supports MISRA C 2012 AMD2 Rule 21.21  
 Supports MISRA C 2012 AMD3 Rule 21.24  
 Supports MISRA C 2012 Dir 4.3  
 Supports MISRA C 2012 Dir 4.12  
 Supports MISRA C 2012 Rule 8.14  
 Supports MISRA C 2012 Rule 21.3  
 Supports MISRA C 2012 Rule 21.5  
 Supports MISRA C 2012 Rule 21.6  
 Supports MISRA C 2012 Rule 21.7  
 Supports MISRA C 2012 Rule 21.8  
 Supports MISRA C 2012 Rule 21.9  
 Supports MISRA C 2012 Rule 21.10  
 Supports MISRA C 2012 Rule 21.12  
 Supports MISRA C++ Rule 17-0-5  
 Supports MISRA C++ Rule 18-0-2  
 Supports MISRA C++ Rule 18-0-3  
 Supports MISRA C++ Rule 18-0-5  
 Supports MISRA C++ Rule 18-2-1  
 Supports MISRA C++ Rule 18-4-1  
 Supports MISRA C++ Rule 19-3-1  
 Supports MISRA C 2004 Rule 2.1  
 Supports MISRA C 2004 Rule 20.4  
 Supports MISRA C 2004 Rule 20.5  
 Supports MISRA C 2004 Rule 20.6  
 Supports MISRA C 2004 Rule 20.7  
 Supports MISRA C 2004 Rule 20.8  
 Supports MISRA C 2004 Rule 20.10  
 Supports MISRA C 2004 Rule 20.11  
 Supports MISRA C 2004 Rule 20.12  
 Supports CWE-1341 - *Multiple Releases of Same Resource or Handle*

**587** **warning** predicate '*string*' can be pre-determined and always evaluates to *true/false*  
 The predicate, identified by *string*, cannot possibly be other than what is indicated by the message. For example:

```

unsigned u; ...
if( (u & 0x10) == 0x11 ) ...

```

would be greeted with the message that '==' always evaluates to 'false'.

See [Precision, Viable Bit Patterns, and Representable Values](#) for more information.

Supports AUTOSAR17 Rule M0-1-9  
 Supports AUTOSAR19 Rule M0-1-9  
 Supports MISRA C++ Rule 0-1-9

**592** **warning** non-literal format specifier used without arguments  
 A printf/scanf style function received a non-literal format specifier without trailing arguments. For example:

```

char msg[100];
...
printf( msg );

```

This can easily be rewritten to the relatively safe:

```
char msg[100];
...
printf( "%s", msg );
```

The danger lies in the fact that `msg` can contain hidden format codes. If `msg` is read from user input, then in the first example, a naive user could cause a glitch or a crash and a malicious user might exploit this to undermine system security. Since the unsafe form can easily be transformed into the safe form, the latter should always be used.

**Supports CERT C FIO30-C** - *Exclude user input from format strings*

**Supports CWE-20** - *Improper Input Validation*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

### 593 custodial pointer *symbol* possibly not *string* nor returned

warning

This is the 'possible' version of message 429. A pointer of `auto` storage class was allocated storage and not all paths leading to a `return` statement or to the end of the function contained either a `free` or a `return` of the pointer. Hence there is a potential memory leak. For example:

```
void f(int n) {
    int *p = new int;
    if (n) delete p;
} //message 593
```

In this example an allocation is made and, if `n` is 0, no `delete` will have been made.

Please see message 429 for an explanation of "custodial" and ways of regulating when pointer variables retain custody of allocations.

**Supports CWE-772** - *Missing Release of Resource after Effective Lifetime*

### 597 suspicious use of unary operator could be confused for compound assignment (*string*)

warning

A construct such as:

```
a -= b;
```

or

```
a =+ b;
```

which is suspect: did the programmer intend to use the `-=`/`+=` compound assignment operator? The message is only issued when there is no space between the `=` and the `-/+` and when there is a space between the `-/+` and its operand. '*string*' contains the form of compound assignment that the expression may be confused for.

### 598 excessive shift value (precision '*integer*' shifted by '*integer*')

warning

A quantity is being shifted to the left by a value greater than or equal to the precision of that quantity or by a negative value. For example,

```
i << 32
```

will elicit this message if `i` is typed `int` and where `int` is 32 bits wide or less (the usual case). Such shift results in undefined behavior. To suppress the message you may cast the shifted quantity to a type whose length is at least the length of the shift value.

**Supports CWE-1335** - *Incorrect Bitwise Shift of Integer*

### 599 cannot open file matching wild card pattern '*string*'

warning

A wild card pattern was used where the name of a file was expected but there were no files found that match the given pattern so it will be ignored. '*string*' contains the offending pattern.



**601 expected a type, int assumed**  
**warning** A declaration did not have an explicit type. `int` was assumed. Was this a mistake? This could easily happen if an intended comma was replaced by a semicolon. For example, if instead of typing:

```
double    radius,
          diameter;
```

the programmer had typed:

```
double    radius;
          diameter;
```

this message would be raised.

**Supports CERT C DCL31-C - Declare identifiers before using them**

**Supports MISRA C 2012 Rule 8.1**

**Supports MISRA C 2004 Rule 8.2**

**602 '/' within block comment**  
**warning** The sequence `/*` was found within a comment. Was this deliberate? Or was a comment end inadvertently omitted? If you want PC-lint Plus to recognize nested comments you should set the Nested Comment flag using the `+fnc` option. Then this warning will not be issued. If it is your practice to use the sequence:

```
/*
/*          */
```

then use `-e602`.

**Supports AUTOSAR19 Rule M2-7-1**

**Supports CERT C MSC04-C - Use comments consistently and in a readable fashion**

**Supports MISRA C 2012 Dir 4.4**

**Supports MISRA C 2012 Rule 3.1**

**Supports MISRA C++ Rule 2-7-1**

**Supports MISRA C 2004 Rule 2.3**

**Supports MISRA C 2004 Rule 2.4**

**603 argument to parameter of type pointer to const may be a pointer to uninitialized memory**  
**warning** The address of the named symbol is being passed to a function where the corresponding parameter is declared as pointer to `const`. This implies that the function will not modify the object. If this is the case then the original object should have been initialized sometime earlier.

**Supports CERT C EXP33-C - Do not read uninitialized memory**

**Supports MISRA C 2012 Rule 9.1**

**Supports CWE-457 - Use of Uninitialized Variable**

**Supports CWE-758 - Reliance on Undefined, Unspecified, or Implementation-Defined Behavior**

**Supports CWE-908 - Use of Uninitialized Resource**

**604 returning address of auto variable *symbol***  
**warning** The address of the named symbol is being passed back by a function. Since the object is an `auto` and since the duration of an `auto` is not guaranteed past the `return`, this is most likely an error. You may want to copy the value into a global variable and pass back the address of the global or you might consider having the caller pass an address of one of its own variables to the callee.

**Supports AUTOSAR17 Rule M7-5-1**

**Supports AUTOSAR19 Rule M7-5-1**

**Supports CERT C DCL30-C - Declare objects with appropriate storage durations**

**Supports MISRA C 2012 Rule 1.3**

Supports MISRA C 2012 Rule 18.6

Supports MISRA C++ Rule 7-5-1

Supports MISRA C 2004 Rule 1.2

Supports MISRA C 2004 Rule 17.6

Supports CWE-562 - Return of Stack Variable Address

**605** **pointee implicitly gains/loses const/volatile qualifier in conversion from type to type (context)**  
**warning** This warning is typically caused by assigning a (pointer to `const`) to an ordinary pointer. For example:

```
int *p;
const int *q;
p = q;      /* 605 */
```

The message will be inhibited if a cast is used as in:

```
p = (int *) q;
```

An increase in capability is indicated because the `const` pointed to by `q` can now be modified through `p`. This message can be given for the `volatile` qualifier as well as the `const` qualifier and may be given for arbitrary pointer depths (pointers to pointers, pointers to arrays, etc.).

If the number of pointer levels exceeds one, things get murky in a hurry. For example:

```
const char ** ppc;
char ** pp;
pp = ppc;      /* 605 - clearly not safe */
ppc = pp;      /* 605 - looks safe but it's not */
```

The problem is that after the above assignment, a pointer to a `const char` can be assigned indirectly through `ppc` and accessed through `pp`, which can then modify the `const char`.

The message speaks of an "increase in capability" in assigning to `ppc`, which seems counter intuitive because the indirect pointer has less capability. However, assigning the pointer does not destroy the old one and the combination of the two pointers represents a net increase in capability.

The message may also be given for function pointer assignments when the prototype of one function contains a pointer of higher capability than a corresponding pointer in another prototype. There is a curious inversion here whereby a prototype of lower capability translates into a function of greater trust and hence greater capability (a Trojan Horse). For example, let

```
void warrior( char * );
```

be a function that destroys its argument. Consider the function:

```
void Troy( void (*horse)(const char *) );\
```

`Troy()` will call `horse()` with an argument that it considers precious (i.e. not to be modified) believing the `horse()` will do no harm. Before compilers knew better and believing that adding in a `const` to the destination never hurt anything, earlier compilers allowed the Greeks to pass `warrior()` to `Troy` and the rest, as they say, is history.

Supports MISRA C++ Rule 9-3-1

**606** **non-ANSI escape sequence: '\string'**  
**warning** An escape sequence occurred, within a character or string literal, that was not on the approved list, which is:

```
\' \" \? \\a \b \f \n \r \t \v
octal-digits xhex-digits
```

Supports AUTOSAR17 Rule A2-14-1

Supports AUTOSAR19 Rule A2-13-1

Supports MISRA C 2012 Rule 1.3

Supports MISRA C++ Rule 2-13-1

Supports MISRA C 2004 Rule 4.1

- 608** **assigning to array parameter *symbol***  
**warning** An assignment is being made to a parameter that is typed array. For the purpose of the assignment, the parameter is regarded as a pointer. Normally such parameters are typed as pointers rather than arrays. However if this is your coding style you should suppress this message.
- 611** **cast between pointer to function type *type* and pointer to object type *type***  
**warning** Either a pointer to a function is being cast to a pointer to an object or vice versa. This is regarded as questionable by the language standards. If this is not a user error, suppress this warning.  
 Supports AUTOSAR17 Rule M5-2-6  
 Supports AUTOSAR19 Rule M5-2-6  
 Supports MISRA C++ Rule 5-2-6
- 612** **declaration does not declare anything**  
**warning** A declaration contained just a storage class and a type. This is almost certainly an error since the only time a type without a declarator makes sense is in the case of a **struct**, **union** or **enum** but in that case you wouldn't use a storage class.  
 Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*  
 Supports CWE-561 - *Dead Code*
- 613** **potential use of null pointer *symbol***  
**warning** From information gleaned from earlier statements, it is possible that a null pointer (a pointer whose value is 0) can be used in a context where null pointers are inappropriate. Information leading to this determination is provided as a series of supplemental messages. See also message [413](#).  
 Supports AUTOSAR19 Rule A5-3-2  
 Supports CERT C EXP34-C - *Do not dereference null pointers*  
 Supports CERT C ARR00-C - *Understand how arrays work*  
 Supports CERT C ARR30-C - *Do not form or use out-of-bounds pointers or array subscripts*  
 Supports CERT C MEM11-C - *Do not assume infinite heap space*  
 Supports CERT C API00-C - *Functions should validate their parameters*  
 Supports CERT C MSC19-C - *For functions that return an array, prefer returning an empty array over a null value*  
 Supports CERT C POS54-C - *Detect and handle POSIX library errors*  
 Supports CWE-119 - *Improper Restriction of Operations within the Bounds of a Memory Buffer*  
 Supports CWE-123 - *Write-what-where Condition*  
 Supports CWE-125 - *Out-of-bounds Read*  
 Supports CWE-127 - *Buffer Under-read*  
 Supports CWE-129 - *Improper Validation of Array Index*  
 Supports CWE-252 - *Unchecked Return Value*  
 Supports CWE-253 - *Incorrect Check of Function Return Value*  
 Supports CWE-391 - *Unchecked Error Condition*  
 Supports CWE-476 - *NULL Pointer Dereference*  
 Supports CWE-690 - *Unchecked Return Value to NULL Pointer Dereference*  
 Supports CWE-786 - *Access of Memory Location Before Start of Buffer*  
 Supports CWE-787 - *Out-of-bounds Write*

**614 auto aggregate initializer not constant**  
**warning** Prior to C99, auto aggregate initialization could consist only of constant expressions. This message is only issued in C89/C90 mode. See also message [446](#).

**615 auto aggregate initializer has side effects**  
**warning** This warning is similar to 614. Auto aggregates (arrays, structures and possibly unions) are normally initialized by a collection of constant expressions without side-effects. If your compiler supports side-effects in the initializers of aggregate, you may want to suppress this message. This message is only issued in C89/C90 mode.

**616 control flow falls through to next case without an intervening comment**  
**warning** It is possible for flow of control to fall into a case statement or a `default` statement from above. Was this deliberate or did the programmer forget to insert a `break` statement? If this was deliberate then place a comment immediately before the statement that was flagged as in:

```
case 'a': a=0;
    /* fall through */
case 'b': a++;
```

Note that the message will not be given for a `case` that merely follows another `case` without an intervening statement. Also, there must actually be a possibility for flow to occur from above. See also message [825](#) and option `-fallthrough`.

**Supports CERT C MSC17-C** - *Finish every set of statements associated with a case label with a break statement*

**Supports MISRA C 2012 Rule 16.1**

**Supports MISRA C 2012 Rule 16.3**

**Supports CWE-484** - *Omitted Break Statement in Switch*

**618 storage class specified after a type**  
**warning** A storage class specifier (`static`, `extern`, `typedef`, `register` or `auto`) was found after a type was specified. This is legal but unusual. Either place the storage class specifier before the type or suppress this message.

**Supports AUTOSAR17 Rule A7-1-8**

**Supports AUTOSAR19 Rule A7-1-8**

**620 suspicious constant (L or one?)**  
**warning** A constant ended in a lower-case letter 'l'. Was this intended to be a one? The two characters look very similar. To avoid misinterpretations, use the upper-case letter 'L'.

**Supports CERT C DCL16-C** - *Use "L," not "l," to indicate a long value*

**Supports MISRA C 2012 Rule 7.3**

**Supports CWE-398** - *Indicator of Poor Code Quality*

**621 identifier clash (*string*): *string* '*string*' clashes with *string* '*string*'**  
**warning** The `-idlen` option can be used to specify the number of *significant characters* in *external*, *preprocessor*, and *preprocessor* names. Names that are not unique within the initial characters specified by this option are said to "clash" and are reported by this message.

For the purpose of this message, identifiers are classified as *external* (function and variable names with external linkage), *preprocessor* (macro names and macro parameter names), and *compiler* (all other identifiers, including those with internal and no linkage, such as fields, tags, enumeration constants, typedefs, labels, etc). The type of clash is reported by the first *string* parameter. The possible values of this parameter and the cases in which the clashes are reported are detailed in the following list.

- **field vs field**  
The names of two fields clash within the same structure or union.
- **tag vs tag**  
The names of two struct, union, or enum tags clash within a single translation unit.
- **label vs label**  
The names of two labels clash within a single function.
- **internal vs internal, same scope**  
Two *compiler* identifiers clash in the same scope.
- **internal vs external, same scope**  
A *compiler* identifier clashes with an *external* identifier in the same scope.
- **external vs internal, same scope**  
An *external* identifier clashes with a *compiler* identifier in the same scope.
- **internal vs internal, enclosing scope**  
A *compiler* identifier clashes with another *compiler* identifier in an enclosing scope.
- **internal vs external, enclosing scope**  
A *compiler* identifier clashes with an *external* identifier in an enclosing scope.
- **external vs internal, enclosing scope**  
An *external* identifier clashes with a *compiler* identifier in an enclosing scope.
- **external vs external**  
Two *external* identifiers clash (anywhere in the analyzed program).
- **macro vs macro**  
The names of two macros in the same translation unit clash.
- **macro vs macro parameter**  
The name of a macro clashes with the name of a macro parameter of a currently defined macro.
- **macro parameter vs macro parameter**  
The name of a macro parameter clashes with the name of a parameter of the same macro.

Fields, tags, and labels each exist in their own name spaces and thus never clash with identifiers in other name spaces. *Internal* here refers to *compiler* identifiers that are not field, tag, or label identifiers. For clashes between internal and external names, the number of significant characters for *compiler* identifiers is used to determine a clash. Clashes between two *external* identifiers are reported regardless of scope. External identifiers in separate modules that clash are reported during global wrapup.

This message is not issued for C++ modules (all characters in identifier names are significant and case-sensitive in C++) or for identifiers with identical spelling.

**Supports CERT C DCL23-C** - *Guarantee that mutually visible identifiers are unique*

**Supports CERT C DCL40-C** - *Do not create incompatible declarations of the same function or object*

**Supports MISRA C 2012 Rule 5.1**

**Supports MISRA C 2012 Rule 5.2**

**Supports MISRA C 2012 Rule 5.3**

**Supports MISRA C 2012 Rule 5.4**

## 629 **'static' function declaration at block scope is non standard**

**warning**

A **static** storage class specifier was found for a function declaration within a function. The **static** storage class specifier is permitted only for functions in declarations that have file scope (i.e., outside any function). Either move the declaration outside the function or change **static** to **extern**; if the second choice is made, make sure that a **static** declaration at file scope also exists before the **extern** declaration.

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2004 Rule 1.2**

- 631 tag *symbol* defined differently at *location***  
**warning** The `struct`, `union` or `enum` tag *symbol* was defined differently in different scopes. This is not necessarily an error since C permits the redefinition, but it can be a source of subtle error. It is not generally a programming practice to be recommended. This message is suppressed for unit checkout (`-unit_check` option).  
 Supports MISRA C 2012 Rule 5.7
- 632 strong type mismatch: assigning '*strong-type*' to '*strong-type*' in context '*context*'**  
**warning** An assignment (or implied assignment, *context* indicates which), violates a Strong type check as requested by a `-strong(A...)` option. See Chapter 7 Strong Types.
- 633 strong type mismatch: extracting '*strong-type*' into '*strong-type*' in context '*context*'**  
**warning** An assignment (or implied assignment, *context* indicates which), violates a Strong type check as requested by a `-strong(X...)` option. See Chapter 7 Strong Types.
- 634 strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**  
**warning** An equality operation (`==` or `!=`) or a conditional operation (`? :`) violates a Strong type check as requested by a `-strong(J...)` option. This message would have been suppressed using flags "Je". See Chapter 7 Strong Types.
- 635 changing the *parent/index* type of '*strong-type*' from '*strong-type*' to '*strong-type*'**  
**warning** The strong parent of the *symbol* is being reset. This is being done with a `-parent` option or by a `typedef`. Note that this may not necessarily be an error; you are being alerted to the fact that the old link is being erased. See Chapter 7 Strong Types.
- 636 strong type difference: pointees are '*strong-type*' and '*strong-type*'**  
**warning** Pointers are being compared and there is a strong type clash below the first level. For example,
- ```

/*lint -strong(J, INT) */
typedef int INT;
INT *p;  int *q;

if( p == q ) /* Warning 636 */

```
- will elicit this warning. This message would have been suppressed using flags "Je" or "Jr" or both. See Chapter 7 Strong Types.
- 637 strong type mismatch: '*strong-type*' is not an acceptable index type for '*strong-type*' (expected '*strong-type*')**  
**warning** This is the message you receive when an inconsistency with the `-index` option is recognized. A subscript is not the stipulated type (the first type mentioned in the message) nor equivalent to it within the hierarchy of types. See Chapter 7 Strong Types and also `+fhx`.
- 638 strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**  
**warning** A relational operation (`>=` `<=` `>` `<`) violates a Strong type check as requested by a `-strong(J...)` option. This message would have been suppressed using flags "Jr". See Chapter 7 Strong Types.

**639 warning** **strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**  
 A binary operation other than an equality or a relational operation violates a Strong type check as requested by a **-strong(J...)** option. This message would have been suppressed using flags "Jo". See Chapter 7 [Strong Types](#).

**640 warning** **strong type mismatch: '*strong-type*' is not an acceptable boolean type for parameter '*keyword*' statement (expected '*strong-type*')**  
 A Boolean context expected a type specified by a **-strong(B...)** option. See Chapter 7 [Strong Types](#).

**641 warning** **implicit conversion of enum *symbol* to integral type *type***  
 An enumeration type was used in a context that required a computation such as an argument to an arithmetic operator or was compared with an integral argument. This warning will be suppressed if you use the integer model of enumeration (**+fie**) but you will lose some valuable type-checking in doing so. An intermediate policy is to simply turn off this warning. Assignment of **int** to **enum** will still be caught.

This warning is not issued for a tagless **enum** without variables. For example

```
enum {false,true};
```

This cannot be used as a separate type. PC-lint Plus recognizes this and treats **false** and **true** as arithmetic constants.

**644 warning** **potentially using an uninitialized value**  
 An auto variable was not necessarily assigned a value before use.  
 Supports CERT C EXP33-C - *Do not read uninitialized memory*  
 Supports MISRA C 2012 Rule 9.1  
 Supports MISRA C 2004 Rule 9.1  
 Supports CWE-456 - *Missing Initialization of a Variable*  
 Supports CWE-457 - *Use of Uninitialized Variable*  
 Supports CWE-758 - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*  
 Supports CWE-908 - *Use of Uninitialized Resource*

**646 warning** **'*string*' within '*string*' loop; may have been misplaced**  
 A case or default statement was found within a for, do, or while loop. Was this intentional? At the very least, this reflects poor programming style.  
 Supports CERT C MSC20-C - *Do not use a switch statement to transfer control into a complex block*  
 Supports MISRA C++ Rule 15-0-3

**647 warning** **possible truncation before conversion from *type* to *type***  
 This message is issued when it appears that there may have been an unintended loss of information during an operation involving **int** or **unsigned int** the result of which is later converted to **long**. It is issued only for systems in which **int** is smaller than **long**. For example:

```
(long) (n << 8)
```

might elicit this message if **n** is **unsigned int**, whereas

```
(long) n << 8
```

would not. In the first case, the shift is done at **int** precision and the high order 8 bits are lost even though there is a subsequent conversion to a type that might hold all the bits. In the second case, the shifted bits are retained.

The operations that are scrutinized and reported upon by this message are: shift left, multiplication, and bit-wise complementation. Addition and subtraction are covered by Informational message [776](#).

The conversion to `long` may be done explicitly with a cast as shown or implicitly via assignment, return, argument passing or initialization.

The message can be suppressed by casting. You may cast one of the operands so that the operation is done in full precision as is given by the second example above. Alternatively, if you decide there is really no problem here (for now or in the future), you may cast the result of the operation to some form of `int`. For example, you might write:

```
(long) (unsigned) (n << 8)
```

In this way PC-lint Plus will know you are aware of and approve of the truncation.

#### **648 overflow in computing constant for operator '*operator*'**

**warning** Arithmetic overflow was detected while computing a constant expression. For example, if `int` is 16 bits then `200 * 200` will result in an overflow.

To suppress this message for particular constant operations you may have to supply explicit truncation. For example, if you want to obtain the low order 8 bits of the integer 20000 into the high byte of a 16-bit `int`, shifting left would cause this warning. However, truncating first and then shifting would be OK. The following code illustrates this where `int` is 16 bits.

```
20000u << 8;          /* 648 */
(0xFF & 20000u) << 8; /* OK */
```

If you truncate with a cast you may make a signed expression out of an unsigned. For example, the following receives a warning (for 16 bit `int`).

```
(unsigned char) 0xFFFu << 8;    /* 648 */
```

because the `unsigned char` is promoted to `int` before shifting. The resulting quantity is actually negative. You would need to revive the `unsigned` nature of the expression with

```
(unsigned) (unsigned char) 0xFFFF << 8;    /* OK */
```

**Supports CERT C INT08-C - Verify that all integer values are in range**

#### **649 right shifting a negative constant expression has implementation defined behavior**

**warning** During the evaluation of a constant expression, a negative integer was shifted right causing sign fill of vacated positions. If this is what is intended, suppress this error, but be aware that sign fill is implementation-dependent.

#### **650 constant '*integer*' out of range for operator '*string*'**

**warning** In a comparison operator or equality test (or implied equality test as for a `case` statement), a constant operand was used in a way that is not appropriate for the constraints on the value of the other operand. For example, if 300 is compared against a `char` variable, this warning will be issued. Moreover, if `char` is signed (and 8 bits) you will get this message if you compare against an integer greater than 127. The problem can be fixed with a cast. For example:

```
if( ch == 0xFF ) ...
if( (unsigned char) ch == 0xFF ) ...
```

If `char` is signed (`+fcs` has not been set) the first receives a warning and can never succeed. The second suppresses the warning and corrects the bug.



PC-lint Plus will take into account the limited precision of some operands such as bit-fields and enumerated types. Also, PC-lint Plus will take advantage of computations that limit the precision of an operand. For example,

```
if( (n & 0xFF) >> 4 == 16 ) ...}
```

will receive this warning because the left-hand side is limited to 4 bits of precision.

See [Precision, Viable Bit Patterns, and Representable Values](#) for more information. See also message [2650](#) for constants that are out of range for only part of a compound comparison operator.

**Supports CERT C INT08-C** - *Verify that all integer values are in range*

**Supports MISRA C 2012 Rule 14.3**

**Supports MISRA C 2004 Rule 13.7**

**651 inconsistent bracing in aggregate initialization**  
**warning** An initializer for a complex aggregate is being processed that contains some subaggregates that are bracketed and some that are not. ANSI/ISO recommends either "minimally bracketed" initializers in which there are no interior braces or "fully bracketed" initializers in which all interior aggregates are bracketed.

**652 #define of macro 'string' with same name as previously declared symbol *symbol***  
**warning** A macro is being defined for a symbol that had previously been declared. For example:

```
int n;
#define n N
```

will draw this complaint. Prior symbols checked are local and global variables, functions and **typedef** symbols, and **struct**, **union** and **enum** tags. Not checked are **struct** and **union** members.

**653 result of integer division being converted to *type***  
**warning** When two integers are divided and assigned to a floating point variable the fraction portion is lost. For example, although

```
double x = 5 / 2;
```

appears to assign 2.5 to **x** it actually assigns 2.0. To make sure you do not lose the fraction, cast at least one of the operands to a floating point type. If you really wish to do the truncation, cast the resulting divide to an integral (**int** or **long**) before assigning to the floating point variable.

**Supports CERT C FLP06-C** - *Convert integers to floating point for floating-point operations*

**654 option '*option*' is obsolete; *detail***  
**warning** The specified *option* is obsolete and should no longer be used. The *detail* parameter contains additional information such as further explanation or alternatives.

**655 bitwise operation uses compatible enums (of type *type*)**  
**warning** A bit-wise operator (one of '**|**', '**&**' or '**^**') is used to combine two compatible enumerations. The type of the result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

**656 arithmetic operation uses compatible enums (of type *type*)**  
**warning** An arithmetic operator (one of '**+**', or '**-**') is used to combine two compatible enumerations. The type of the result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

**657 unusual (nonportable) anonymous struct or union**  
**warning** A `struct` or `union` declaration without a declarator was taken to be anonymous. However, the anonymous `union` supported by C++ and other dialects of C require untagged unions. Tagged unions and tagged or untagged structs are rarely supported, as anonymous.

**658 anonymous union assumed (use flag `+fan`)**  
**warning** A union without a declarator was found. Was this an attempt to define an anonymous `union`? If so, anonymous unions should be activated with the `+fan` flag. This flag is activated automatically for C++.

**660 option '`string`' requests removing an extent that is not on the list**  
**warning** A number of options use the '-' prefix to remove and the '+' prefix to add elements to a list. For example to add (the most unusual) extension `.C++` to designate C++ processing of files bearing that extension, a programmer should employ the option:

```
+cpp(.C++)
```

However, if a leading '-' is employed (a natural mistake) this warning will be emitted.

**661 potential out of bounds pointer access: excess of *integer* byte(s)**  
**warning** An out-of-bounds pointer may have been accessed. See message [415](#) for a description of the *integer* parameter. For example:

```
int a[10];
if( n <= 10 ) a[n] = 0;
```

Here the programmer presumably should have written `n < 10`. This message is similar to message [415](#) but differs from it by the degree of probability. See Chapter [8 Value Tracking](#).

**Supports AUTOSAR17 Rule M5-0-16**

**Supports AUTOSAR17 Rule A5-2-5**

**Supports AUTOSAR19 Rule M5-0-16**

**Supports AUTOSAR19 Rule A5-2-5**

**Supports CERT C ARR30-C - Do not form or use out-of-bounds pointers or array subscripts**

**Supports CERT C MSC19-C - For functions that return an array, prefer returning an empty array over a null value**

**Supports MISRA C 2012 Rule 18.1**

**Supports MISRA C++ Rule 5-0-16**

**Supports CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer**

**Supports CWE-123 - Write-what-where Condition**

**Supports CWE-125 - Out-of-bounds Read**

**Supports CWE-126 - Buffer Over-read**

**Supports CWE-127 - Buffer Under-read**

**Supports CWE-129 - Improper Validation of Array Index**

**Supports CWE-786 - Access of Memory Location Before Start of Buffer**

**Supports CWE-787 - Out-of-bounds Write**

**662 possibly creating out-of-bounds pointer: excess of *integer* byte(s)**  
**warning** An out-of-bounds pointer may have been created. See message [415](#) for a description of the *integer* parameter. For example:

```
int a[10];
if( n <= 20 ) f( a + n );
```

Here, it appears as though an illicit pointer is being created, but PC-lint Plus cannot be certain. See also message 416 and Chapter 8 [Value Tracking](#).

**Supports AUTOSAR17 Rule M5-0-16**

**Supports AUTOSAR17 Rule A5-2-5**

**Supports AUTOSAR19 Rule M5-0-16**

**Supports AUTOSAR19 Rule A5-2-5**

**Supports CERT C ARR30-C - Do not form or use out-of-bounds pointers or array subscripts**

**Supports CERT C MSC19-C - For functions that return an array, prefer returning an empty array over a null value**

**Supports MISRA C 2012 Rule 18.1**

**Supports MISRA C++ Rule 5-0-16**

**Supports CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer**

**Supports CWE-123 - Write-what-where Condition**

**Supports CWE-125 - Out-of-bounds Read**

**Supports CWE-126 - Buffer Over-read**

**Supports CWE-127 - Buffer Under-read**

**Supports CWE-129 - Improper Validation of Array Index**

**Supports CWE-786 - Access of Memory Location Before Start of Buffer**

**Supports CWE-787 - Out-of-bounds Write**

**663** **array-to-pointer decay causes indirection through first element**  
**warning** This warning occurs in the following kind of situation:

```
struct x { int a; } y[2];
... y->a ...
```

Here, the programmer forgot to index the array but the error normally goes undetected because the array reference is automatically and implicitly converted to a pointer to the first element of the array. If you really mean to access the first element use `y[0].a`

**664** **left hand side of logical operator contains call to function that does not return**  
**warning** An exiting function was found on the left hand side of an operator implying that the right hand side would never be executed. For example:

```
if( (exit(0), n == 0) || n > 2 ) ...
```

Since the `exit` function does not return, control can never flow to the right hand operator.

**665** **unparenthesized parameter *integer* in macro '*string*' is passed an expression**  
**warning** An expression was passed to a macro parameter that was not parenthesized. For example:

```
#define mult(a,b) (a*b)
... mult( 100, 4 + 10 )
```

Here the programmer is beguiled into thinking that the `4+10` is taken as a quantity to be multiplied by 100 but instead results in: `100*4+10`, which is quite different. The recommended remedy ([8, Section 19.4]) is to parenthesize such parameters as in:

```
#define mult(a,b) ((a)*(b))
```

The message is not arbitrarily given for any unparenthesized parameter but only when the actual macro argument sufficiently resembles an expression and the expression involves binary operators. The priority of the operator is not considered except that it must have lower priority than the unary operators. The message is not issued at the point of macro definition because it may not be appropriate to parenthesize the parameter. For example, the following macro expects that an operator will be passed as argument. It would be an error to enclose `op` in parentheses.

```
#define check(x,op,y) if( ((x) op (y)) == 0 ) print( ... )
```

Supports MISRA C 2012 Rule 20.7

- 666** **expression with side effects passed to repeated parameter *integer* of macro '*string*'**  
**warning** A repeated parameter within a macro was passed an argument with side-effects. For example:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))

... ABS( n++ )
```

Although the ABS macro is correctly defined to specify the absolute value of its argument, the repeated use of the parameter *x* implies a repeated evaluation of the actual argument *n++*. This results in two increments to the variable *n*. [8, Section 19.6] Any expression containing a function call is also considered to have side-effects.

Supports CERT C PRE31-C - *Avoid side effects in arguments to unsafe macros*

- 668** **possibly passing null pointer to function *symbol*, *context***  
**warning** A NULL pointer is possibly being passed to a function identified by *symbol*. The argument in question is given by *context*. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option **-function** or **-sem**. See Sections [9.1 Function Mimicry \(-function\)](#), [9.2 Semantic Specifications](#) and [8 Value Tracking](#).

Supports CERT C EXP34-C - *Do not dereference null pointers*

Supports CERT C API00-C - *Functions should validate their parameters*

Supports CERT C MSC19-C - *For functions that return an array, prefer returning an empty array over a null value*

Supports CWE-476 - *NULL Pointer Dereference*

Supports CWE-690 - *Unchecked Return Value to NULL Pointer Dereference*

- 669** **possible data overrun for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**  
**warning** This message is for data transfer functions such as `memcpy`, `strcpy`, `fgets`, etc. when the size indicated by the first cited argument (or arguments) can possibly exceed the size of the buffer area cited by the second. The message may also be issued for user functions via the **-function** or **-sem** option. See Sections [9.1 Function Mimicry \(-function\)](#), [9.2 Semantic Specifications](#) and [Chapter 8 Value Tracking](#).

Supports CERT C ENV01-C - *Do not make assumptions about the size of an environment variable*

Supports CERT C MSC19-C - *For functions that return an array, prefer returning an empty array over a null value*

- 670** **possible access beyond array for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**  
**warning** This message is issued for several library functions (such as `fwrite`, `memcpy`, etc.) wherein there is a possible attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified. The function is specified by *symbol* and the arguments are identified by argument number. See Sections [9.1 Function Mimicry \(-function\)](#), [9.2 Semantic Specifications](#) and [Chapter 8 Value Tracking](#).

Supports CERT C MSC19-C - *For functions that return an array, prefer returning an empty array over a null value*

Supports CWE-119 - *Improper Restriction of Operations within the Bounds of a Memory Buffer*

- 671** **possibly passing to function *symbol* a negative value (*string*) *context***  
**warning** An integral value that may possibly be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (*symbol*), the questionable value (*integer*) and the argument number (*context*). The function may be a standard library function designed to accept only positive values such as `malloc` or `memcpy` (third argument), or may have been identified

by the user as such through the `-function` or `-sem` options. See message 422 for an example and further explanation.

**Supports** **CWE-687** - *Function Call With Incorrectly Specified Argument Value*

**672** **assignment to custodial pointer *symbol* possibly creates memory leak**

**warning**

An assignment was made to a pointer variable (designated by *symbol*), which may already be holding the address of an allocated object that had not been freed. The allocation of memory, which is not freed, is considered a 'memory leak'. The memory leak is considered 'possible' because only some lines of flow will result in a leak.

**Supports** **CWE-772** - *Missing Release of Resource after Effective Lifetime*

**673** ***string* may not be appropriate for deallocating *string***

**warning**

This message indicates that a deallocation (`delete`, `delete[]`, or `free`) as specified by the first *string* parameter may be inappropriate for the data being freed. The kind of data is described in the second *string* parameter. The wording 'may not' is used to indicate that only some of the lines of flow to the deallocation show data inconsistent with the allocation. See also message 424.

**Supports** **CERT C MEM34-C** - *Only free memory allocated dynamically*

**Supports** **CWE-404** - *Improper Resource Shutdown or Release*

**Supports** **CWE-590** - *Free of Memory not on the Heap*

**Supports** **CWE-762** - *Mismatched Memory Management Routines*

**674** **returning address of auto variable *symbol* through pointer *symbol***

**warning**

The value held by a pointer variable contains the address of an `auto` variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

**Supports** **CERT C DCL30-C** - *Declare objects with appropriate storage durations*

**Supports** **CWE-562** - *Return of Stack Variable Address*

**675** **no prior semantics associated with '*name*' in option '*string*'**

**warning**

The `-function` option is used to transfer semantics from its first argument to subsequent arguments. However it was found that the first argument *name* did not have semantics.

**676** **possibly indexing before the beginning of an allocation**

**warning**

An integer whose value was possibly negative was added to an array or to a pointer to an allocated area (allocated by `malloc`, `operator new`, etc.). This message is not given for pointers whose origin is unknown since a negative subscript is in general legal.

**Supports** **CERT C ARR30-C** - *Do not form or use out-of-bounds pointers or array subscripts*

**Supports** **MISRA C 2012 Rule 18.1**

**Supports** **CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*

**Supports** **CWE-123** - *Write-what-where Condition*

**Supports** **CWE-124** - *Buffer Underwrite ('Buffer Underflow')*

**Supports** **CWE-125** - *Out-of-bounds Read*

**Supports** **CWE-126** - *Buffer Over-read*

**Supports** **CWE-127** - *Buffer Under-read*

**Supports** **CWE-129** - *Improper Validation of Array Index*

**Supports** **CWE-786** - *Access of Memory Location Before Start of Buffer*

**Supports** **CWE-787** - *Out-of-bounds Write*

**Supports** **CWE-839** - *Numeric Range Comparison Without Minimum Check*

**677 sizeof used within preprocessor statement**

**warning** Whereas the use of `sizeof` during preprocessing is supported by a number of compilers it is not a part of the ANSI/ISO C or C++ standard. See Section [18.6 Preprocessor sizeof](#).

**678 member *symbol* field length (*integer*) too small for enum precision (*integer*)**

**warning** A bit field was found to be too small to support all the values of an enumeration (that was used as the base of the bit field). For example:

```
enum color { red, green, yellow, blue };
struct abc { enum color c:2; };
```

Here, the message is not given because the four enumeration values of `color` will just fit within 2 bits. However, if one additional color is inserted, Warning 678 will be issued informing the programmer of the undesirable and dangerous condition.

**679 integer operation may be truncated before being combined with a larger pointer type**

**warning** This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals before combining with a pointer whose precision is greater than the integral expression. For example:

```
// Assuming 4-byte ints and 8-bytes pointers (-si4 -sp8)
char *f( char *p, int n, int m ) {
    return p + (n + m); // warning 679
}
```

By the rules of C/C++, the addition `n+m` is performed independently of its context and is done at integer precision. Any overflow is ignored even though the larger precision of the pointer could easily accommodate the overflow. If, on the other hand the expression were: `p+n+m`, which parses as `(p+n)+m`, no warning would be issued.

If the expression were `p+n*m` then, to suppress the warning, a cast is needed. If `long` were the same size as pointers you could use the expression:

```
return p + ((long) n * m);
```

**Supports CERT C INT08-C - Verify that all integer values are in range**

**680 suspicious truncation in arithmetic expression converted to pointer**

**warning** An arithmetic expression was cast to pointer. Moreover, the size of the pointer is greater than the size of the expression. In computing the expression, any overflow would be lost even though the pointer type would be able to accommodate the lost information. To suppress the message, cast one of the operands to an integral type large enough to hold the pointer. Alternatively, if you are sure there is no problem you may cast the expression to an integral type before casting to pointer. See messages [647](#), [776](#), [790](#) and [679](#).

**Supports CERT C INT08-C - Verify that all integer values are in range**

**681 loop is likely not entered**

**warning** The controlling expression for a loop (either the expression within a `while` clause or the second expression within a `for` clause) evaluates initially to 0 and so it appears as though the loop is never entered.

**Supports AUTOSAR17 Rule M0-1-1**

**Supports AUTOSAR19 Rule M0-1-1**

**Supports MISRA C 2012 Rule 2.1**

**Supports MISRA C++ Rule 0-1-1**

Supports MISRA C 2004 Rule 14.1

Supports CWE-561 - Dead Code

**682** **sizeof applied to parameter *symbol* of function *symbol* whose type is a sized array will yield size of *string* instead of *string***

If a parameter is typed as an array it is silently promoted to pointer. Taking the size of such an array will actually yield the size of a pointer. Consider, for example:

```
unsigned f( char a[100] ) { return sizeof(a); }
```

Here it looks as though function `f()` will return the value 100 but it will actually return the size of a pointer, which is usually 4.

Supports CERT C ARR01-C - Do not apply the sizeof operator to a pointer when taking the size of an array

Supports MISRA C 2012 AMD1 Rule 12.5

Supports CWE-467 - Use of sizeof() on a Pointer Type

**683** **function '*string*' #define'd, semantics may be lost**

This message is issued whenever the name of a function with some semantic association is defined as a macro. For example:

```
#define strlen mystrlen
```

will raise this message. The problem is that the semantics defined for `strlen` will then be lost. Consider this message an alert to transfer semantics from `strlen` to `mystrlen`, using `-function(strlen, mystrlen)`. The message will be issued for built-in functions (with built-in semantics) or for user-defined semantics. The message will not be issued if the function is defined to be a function with a similar name but with underscores either appended or prepended or both. For example:

```
#define strlen __strlen
```

will not produce this message. It will produce Info [828](#) instead.

**685** **relational operator '*string*' always evaluates to '*string*'**

The first *string* is one of '>', '>=', '<', or '<=' and identifies the relational operator. The second *string* is one of `true` or `false`. The message is given when an expression is compared to a constant and the precision of the expression indicates that the test will always succeed or always fail. For example,

```
char ch;
...
if( ch >= -128 ) ...
```

In this example, the precision of `char ch` is 8 bits signed (assuming the `fcu` flag has been left in the OFF state) and hence it has a range of values from -128 to 127 inclusive. Hence the test is always `true`.

Note that, technically, `ch` is promoted to `int` before comparing with the constant. For the purpose of this comparison we consider only the underlying precision. As another example, if `u` is an `unsigned int` then

```
if( (u & 0xFF) > 0xFF ) ...
```

will also raise message [685](#) because the expression on the left hand side has an effective precision of 16 bits.

Supports AUTOSAR17 Rule M0-1-1

Supports AUTOSAR17 Rule M0-1-2

Supports AUTOSAR17 Rule M0-1-9

Supports AUTOSAR19 Rule M0-1-1

Supports AUTOSAR19 Rule M0-1-2

Supports AUTOSAR19 Rule M0-1-9

Supports MISRA C 2012 Rule 14.3



Supports MISRA C++ Rule 0-1-1

Supports MISRA C++ Rule 0-1-2

Supports MISRA C++ Rule 0-1-9

Supports MISRA C 2004 Rule 13.7

Supports CWE-570 - *Expression is Always False*

Supports CWE-571 - *Expression is Always True*

## 686 warning option '*option*' is suspicious: *detail*

An option is considered suspicious for one of a variety of reasons. The reason is designated by *detail*. At this writing the following reasons for issuing this message are:

- **unbalanced quotes**  
An option was seen with a quote character that was not balanced within that same option.
- **backtick preceding non-meta character is superfluous and has been dropped**  
A backtick (`) was seen before a character other than a \* or a ?. The use of a backtick in this fashion has no effect.
- **upper case characters within extension '*string*'; these will match lower case when +fff is on; try -fff**  
A file extension involving uppercase letters was seen in a +cpp or +lnt option while the +fff flag was active or the flag became active while there were uppercase extensions registered via +cpp or +lnt. If, for example, you intend for .c to indicate a C module and .C to indicate a C++ module, turning off the fff flag will help avoid unnecessary complaints from PC-lint Plus.
- **extraneous characters following *string***  
One or more characters were seen immediately following a character that is expected to signify the end of an option, such as a closing right parenthesis. While the extraneous characters are ignored, their presence may indicate a typographical error.
- **the likelihood of causing meaningless output**  
An option, such as -elib(\*), -wlib(0), or +fce was seen; this typically hides a problem in the PC-lint Plus configuration. When using a new configuration, it's common for a user to encounter Error messages about Library header code. (This usually does not indicate a problem with library headers.) For example, a misconfiguration of PC-lint Plus preprocessor is by far the most common source of these errors. If you merely suppress basic Syntax Errors (like error 10) and/or Fatal Errors (like error 309), the underlying configuration problem still exists; as a result, PC-lint Plus will fail to parse your code correctly (because your code depends on the aforementioned library code). The output from Lint would then seem illogical and/or meaningless. Therefore, blanket suppression options like this are highly discouraged. Instead, other aspects of the Lint configuration should be modified to make Lint's behavior more similar to that of the compiler at (or, typically, before) the point of Error.
- **it is too late to use -incvar as '*name*' has already been processed as incvar**  
This option (-incvar) is used to specify the name of the environment variable that contains a list of supplementary directories to be searched for headers. This option does not have any effect after this environment variable is processed, which occurs when processing the first module. To have an effect, the option must be moved to before the first module.
- **option has no effect due to zero length zone of transition**  
The -w# or -wlib(#) option was seen with the same warning level of the previously provided -w# or -wlib(#) option. Because the warning level doesn't change, there is no zone of transition and therefore no effect on the message suppression set. For example, in -w1 +e714 -w1, the second -w1 does not



have any effect, in particular, message 714 is not suppressed because there is no zone of transition. If the goal is to suppress all messages except for errors regardless of messages that have been enabled in the meantime, it is necessary to raise the warning level and then lower it, e.g. `-w4 -w1`.

- **modifying the LINT environment variable after startup has no effect; this variable should be set before program startup**  
The `-setenv` option was used to set the LINT environment variable. If this variable is set when PC-lint Plus is started, its contents are processed as options before the command line options are processed. Attempting to set or change the value of this variable after program startup has no effect.
- **`-max_threads` option must appear before first module to have any effect**  
The `-max_threads` option is used to specify the maximum concurrent linting threads to dispatch when performing parallel analysis. This option has no effect when it appears after the first module; move the option to before the first module is referenced to obtain the desired behavior.
- **the size of an incomplete type was requested in a function semantic**  
The use of `@p` was used in a user-defined function return semantic but the pointee return type was not complete at the point of the call. This is suspicious because if the type is incomplete, PC-lint Plus cannot calculate its size from the number of the type's elements. Either use `@P` to specify size in bytes or make a definition of the type visible to PC-lint Plus at the point of the call.
- **the `'-i'` option is used to specify search directories, `'-include'` will add `'nclude'` to the directory search list; use the `-header` option to auto-include files**  
The option `-include` was seen. The likely intention was to cause a particular file to be auto-included as some compilers support this feature with an option of the same name. The PC-lint Plus option to auto-include files is `-header`.
- **the `'-i'` option is used to specify search directories, use the `-header` option to auto-include files**  
An option starting with `-include=` was seen. The likely intention was to cause the file following the `=` to be auto-included as some compilers support this feature with an option of the same name. The PC-lint Plus option to auto-include files is `-header`.
- **path refers to a file rather than a directory**  
The path provided to a `-i` or `-I` option was a valid path but refers to a file rather than a directory.
- **path is not accessible**  
The path provided to a `-i` or `-I` option does not exist or is otherwise inaccessible.
- **path begins with unexpanded tilde prefix**  
The path provided to a `-i` or `-I` option began with a tilde (`~`). This was likely intended to refer to the user's home directory, but it is interpreted literally unless expanded by the shell.
- **path resembles a header file name**  
The argument to a `-i` or `-I` option ended with an extension that is commonly used for header file names such as `.h`, `.hpp`, or `.hxx` and did not correspond to a directory. The argument to a `-i` option should be a directory path, not a file path.
- **language standard cannot be changed while processing a module**  
A `-std` option was encountered within a configuration file opened by `-indirect` or `-subfile` while processing a module. This option will have no effect in this context but would be valid if the

configuration file were used outside of a module. Note that an attempt to change the language standard directly within a module, e.g. using a `-std` option within a comment, is always invalid and error 72 will be reported instead.

- **MISRA diagnostics are no longer issued via message 'msg'**  
A message pattern contained one of 960, 961, 1960, or 1963 which were messages used by PC-lint 9 to issue various MISRA diagnostics. These messages do not exist in PC-lint Plus which instead employs messages specific to individual MISRA guidelines. Options referencing these obsolete messages should be updated.
- **message 'msg' is deprecated and will be removed in a future version**  
The specified deprecated message number was encountered in a message pattern. Deprecated messages will eventually be removed from PC-lint Plus and should not be used. See the description of the cited message number for more specific guidance.
- **the value of the 'flo' flag cannot be changed within a module**  
An attempt was made to change the value of the `flo` flag using `+flo`, `++flo`, `-flo`, or `--flo` within a module. The value of the `flo` flag should only be changed outside a module and these options will have no effect when encountered in a module.
- **option indirectly changes state of 'flo' flag option in a module**  
A `-env_pop` or `-env_restore` option encountered within a module is recalling an option environment in which the state of the `flo` flag option is different. The value of the `flo` flag is not intended to be changed within a module and doing so indirectly likely will not have the desired effect.

#### **687 warning body of 'string' is an unparenthesized comma operator**

A comma operator appeared unbraced and unparenthesized in a statement following an `if`, `else`, `while` or `for` clause. For example:

```
if( n > 0 ) n = 1,
    n = 2;
```

Thus the comma could be mistaken for a semi-colon and hence be the source of subtle bugs.

If the statement is enclosed in curly braces or if the expression is enclosed in parentheses, the message is not issued.

**Supports CWE-398** - *Indicator of Poor Code Quality*

#### **689 warning apparent end of C-style comment ignored**

The pair of characters `'*/'` was found not within a comment. As an example:

```
void f( void*/*comment*/ );
```

This is taken to be the equivalent of:

```
void f( void* );
```

That is, an implied blank is inserted between the `'*'` and the `'/'`. To avoid this message simply place an explicit blank between the two characters.

**Supports CERT C MSC04-C** - *Use comments consistently and in a readable fashion*

#### **691 warning suspicious use of backslash**

The backslash character has been used in a way that may produce unexpected results. Typically this would occur within a macro such as:

```
#define A b \ // comment
```

The coder might be thinking that the macro definition will be continued on to the next line. The standard indicates, however, that the newline will not be dropped in the event of an intervening comment. This should probably be recoded as:

```
#define A b /* comment */ \
```

**Supports CWE-398** - *Indicator of Poor Code Quality*

### 692 decimal character '*string*' follows octal escape sequence '*string*'

**warning**

A *string* was found that contains an '8' or '9' after an octal escape sequence with no more than two octal digits, e.g.

```
"\079"
```

contains two characters: Octal seven (ASCII BEL) followed by '9'. The casual reader of the code (and perhaps even the programmer) could be fooled into thinking this is a single character. If this is what the programmer intended he can also render this as

```
"\07" "9"
```

so that there can be no misunderstanding. On the other hand,

```
"\1238"
```

will not raise a message because it is assumed that the programmer knows that octal escape sequences cannot exceed four characters (including the initial backslash).

**Supports CWE-398** - *Indicator of Poor Code Quality*

### 693 the sequence "*detail*" represents a NUL character followed by the literal string "*detail*"

**warning**

A string was found that looks suspiciously like (but is not) a hexadecimal escape sequence; rather, it is a null character followed by letter "x" followed by some hexadecimal digit, e.g.:

```
"\0x62"
```

was found where the programmer probably meant to type "\x62". If you need precisely this sequence you can use:

```
"\0" "x62"
```

and this warning will not be issued.

**Supports CWE-170** - *Improper Null Termination*

**Supports CWE-463** - *Deletion of Data Structure Sentinel*

### 695 inline function *symbol* declared without storage-class specifier

**warning**

In C99, the result of a call to a function declared with 'inline' but not 'static' or 'extern' is unspecified.

Example: Let the following text represent two translation units:

*module1.c*

```
void f() { }
```

*module2.c*

```
inline void f() { }
void g() { f(); } /* which f() is called? */
```

The C99 Standard dictates that the above call to `f()` from `g()` in `module2.c` may result in the execution of either `f()`.

The programmer may avoid confusion and improve portability by using the keyword `'static'` in addition to `'inline'`. The keyword `'extern'` can also be used along with the `'inline'` to resolve this ambiguity; however, we recommend using `'static'` because, as of this writing, more compilers correctly interpret `'static inline'`.  
**Supports MISRA C 2012 Rule 8.10**

**696 values from *'integer'* to *'integer'* are out of range for operator *'string'***

**warning** The variable is being compared (using one of the 6 comparison operations) with some other expression called the comperand. The variable has a value that is out of the range of values of this comperand. For example consider:

```
void f(unsigned char ch) {
    int n = 1000;
    if (ch < n)      // Message 696
        ...
}
```

Here a message 696 will be issued stating that `n` has a value of 1000 that is out of range because 1000 is not in the set of values that `ch` can hold (assuming default sizes of scalars).

**697 an expression with an integral strong boolean type should be equality-compared only to zero**

**warning** A quasi-boolean value is being compared (using either `!=` or `==`) with a value that is not the literal zero. A quasi-boolean value is any value whose type is a strong boolean type and that could conceivably be something other than zero or one. This is significant because in C, all non-zero values are equally true. Example:

```
/*lint -strong(AJXb, B) */
typedef int B;
#define YES ((B)1)
#define NO  ((B)0)

B f( B a, B b ) {
    B c = ( a == NO);           /*OK, no Warning here*/
    B d = ( a == (b != NO) );   /* Warning 697 for == but not for != */
    B e = ( a == YES );         /* Warning 697 here */
    return d == c;             /* Warning 697 here */
}
```

Note that if `a` and `b` had instead been declared with true boolean types, such as `'bool'` in C++ or `'_Bool'` in C99, this diagnostic would not have been issued.

**Supports CERT C EXP20-C - Perform explicit tests to determine success, true and false, and equality**

**698 in-place realloc of *symbol* could cause a memory leak**

**warning** A statement of the form:

```
v = realloc( v, ... );
```

has been detected. Note the repeated use of the same variable. The problem is that `realloc` can fail to allocate the necessary storage. In so doing it will return `NULL`. But then the original value of `v` is overwritten resulting in a memory leak.

**Supports MISRA C 2012 Rule 22.1**

**701 shift left of signed quantity (*type*)**

**info** Shifts are normally accomplished on **unsigned** operands. This message is only emitted if `sizeof(type) <= sizeof(int)`.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

**702 shift right of signed quantity (*type*)**

**info** Shifts are normally accomplished on **unsigned** operands. Shifting an `int` right is machine dependent (sign fill vs. zero fill). This message is only emitted if `sizeof(type) <= sizeof(int)`.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-197** - *Numeric Truncation Error*

**703 shift left of signed quantity (*type*)**

**info** Shifts are normally accomplished on **unsigned** operands. This message is only emitted if `sizeof(type) > sizeof(int)`.

**704 shift right of signed quantity (*type*)**

**info** Shifts are normally accomplished on **unsigned** operands. Shifting a `long` to the right is machine dependent (sign fill vs. zero fill). This message is only emitted if `sizeof(type) > sizeof(int)`.

**705 format '*string*' specifies type *type* which is nominally inconsistent with argument no. *integer* of *string* type**

The argument corresponding to a conversion specifier in a `printf/scanf` style function was not of the correct type but was the same size as the expected integer type. The format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
extern char * buffer;
sprintf(buffer, "%u", 371);
```

will elicit the message:

```
format '%u' specifies type 'unsigned int' which is nominally
inconsistent with argument no. 3 of type 'int'
```

In addition to differences in signedness of same-sized integers, two types that are the same size and signedness but distinct types are also reported by this message. For example, if `int` and `long` are the same size, passing a `long` argument to `%d` will elicit this message.

**Supports CERT C INT00-C** - *Understand the data model used by your implementation(s)*

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

**Supports CWE-843** - *Access of Resource Using Incompatible Type ('Type Confusion')*

**706 format '*string*' specifies type *type* whose pointee type is nominally inconsistent with argument no. *integer* of *string* type**

The argument corresponding to a conversion specifier in a `printf/scanf` style function was not of the correct type but was a pointer to a type that is the same size as the expected pointee integer type. The

format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
int j;
scanf("%u", &j);
```

will result in the message:

```
format '%u' specifies type 'unsigned int *' whose pointee type
is nominally inconsistent with argument no. 2 of type 'int *'
```

In addition to differences in signedness of same-sized integers, pointers to types that are the same size and signedness but distinct types are also reported by this message. For example, if `int` and `long` are the same size, passing a `long *` argument to `%d` will elicit this message.

**Supports CERT C INT00-C** - *Understand the data model used by your implementation(s)*

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

**Supports CWE-843** - *Access of Resource Using Incompatible Type ('Type Confusion')*

### 707 mixing narrow and *wide/Unicode* string literals in concatenation

**info** The following is an example of a mixing of narrow and wide string literals.

```
const wchar_t *s = "abc" L"def";
```

The concatenation of narrow and wide string literals results in undefined behavior for C90 and C++2003. If your compiler supports such combinations or you use a C/C++ dialect that supports such, you may either suppress this message or consider making the concatenands match.

**Supports CERT C STR10-C** - *Do not concatenate different type of string literals*

**Supports MISRA C++ Rule 2-13-5**

### 708 union initialization

**info** A union was initialized without explicitly specifying which member to initialize. While the C and C++ standards state that the first member of the union is initialized in such cases, other members may not have fully initialized values. For example:

```
union U { int a; int * p; };
U u1 = { 0 };
```

On a system where `int` is 4 bytes and pointers are 8 bytes, the `int` member of `u1` is initialized to 0 but the bytes of `p` that do not overlap with `a` are not initialized, which may come as a surprise, especially since the behavior is dependent on the order in which the union members are declared and on the size of pointers relative to `ints`.

### 709 no intervening module since the last '-pch' option

**info** Two `-pch` options were seen without an intervening module. This is suspicious because the first `-pch` option has no effect in such a case as only one PCH file can be used per module.

### 712 implicit conversion (*context*) from *type* to *type*

**info** An assignment (or implied assignment, see *context*) is being made from a source *type* (the first *type*) to a destination type (the second *type*) and the first *type* is larger than the second *type*. A cast will suppress this message.

**713 implicit conversion (*context*) from *type* to *type***

**info** An assignment (or implied assignment, see *context*) is being made from an unsigned quantity to a signed quantity, that will result in the possible loss of one bit of integral precision, such as converting from **unsigned int** to **int**. A cast will suppress the message.

**714 external symbol *symbol* was defined but not referenced**

**info** The named external variable or external function was defined but not referenced. This message is not issued for library symbols and is suppressed for unit checkout (**-unit\_check** option).

**Supports AUTOSAR17 Rule M0-1-3**

**Supports AUTOSAR17 Rule M0-1-10**

**Supports AUTOSAR19 Rule M0-1-3**

**Supports AUTOSAR19 Rule M0-1-10**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C 2012 AMD4 Rule 2.8**

**Supports MISRA C++ Rule 0-1-3**

**Supports MISRA C++ Rule 0-1-10**

**Supports CWE-561 - Dead Code**

**715 named parameter *symbol* of '*virtual/non-virtual*' function *symbol* not subsequently referenced**

**info** The named formal parameter was not referenced.

**Supports AUTOSAR17 Rule M0-1-11**

**Supports AUTOSAR19 Rule A0-1-4**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports CERT C MSC13-C - Detect and remove unused values**

**Supports MISRA C 2012 Rule 2.7**

**Supports MISRA C++ Rule 0-1-11**

**Supports MISRA C++ Rule 0-1-12**

**Supports CWE-561 - Dead Code**

**716 infinite loop via while**

**info** A construct of the form **while** (1) or **while** (**true**) was found. While this represents a constant value in a context expecting a Boolean — typically reported by message **506** — it may reflect a programming policy to indicate an intentional infinite loop. It is therefore given a separate number and has been placed in the informational category. Some may prefer to denote such loops using **for** ( ; ; ) instead.

**717 monocarpic do-while used to group statements**

**info** A construct of the form **do ... while** (0) or **do ... while** (**false**) was found. While this represents a constant value in a context expecting a Boolean — typically reported by message **506** — it is probably a deliberate attempt on the part of the programmer to encapsulate a sequence of statements into a single statement. It is therefore given a separate number and has been placed in the informational category [8, Section 19.7].

**718 function *symbol* undeclared, assumed to return int**

**info** A function was referenced without having been declared or defined within the current module. Such implicit function declarations were removed in C99 although some compilers still allow them. These implicit function declarations were never allowed in C++ and referencing an undeclared function in a C++ module will instead result in an error. Note that by adding a declaration to another module, you will not suppress this message. It can only be suppressed by placing a declaration within the module being processed.

**Supports CERT C DCL07-C - Include the appropriate type information in function declarators**

**Supports CERT C DCL31-C** - *Declare identifiers before using them*

**Supports MISRA C 2012 Rule 17.3**

**Supports MISRA C 2004 Rule 8.1**

### 719 data argument *integer* not used by format string

**info** The number of data arguments passed to a `printf/scanf` style function was more than what is specified in the format. This message is similar to Warning [558](#), which alerts users to situations in which there were too few arguments for the format. It receives a lighter Informational classification because the additional arguments are simply ignored whereas passing too few arguments results in undefined behavior.

**Supports CERT C DCL10-C** - *Maintain the contract between the writer and caller of variadic functions*

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2004 Rule 1.2**

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-561** - *Dead Code*

**Supports CWE-628** - *Function Call with Incorrectly Specified Arguments*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

### 720 boolean test of assignment

**info** An assignment was found in a context that requires a Boolean (such as the condition of an `if` or `while` statement). This may be legitimate or it could have resulted from a mistaken use of `=` for `==`. If the assignment was intentional, placing additional parenthesis around the assignment (e.g. `if ((a = b))`) will suppress this message.

**Supports AUTOSAR17 Rule M6-2-1**

**Supports AUTOSAR19 Rule M6-2-1**

**Supports CERT C EXP45-C** - *Do not perform assignments in selection statements*

**Supports MISRA C 2012 Rule 13.4**

**Supports MISRA C++ Rule 6-2-1**

**Supports MISRA C 2004 Rule 13.1**

**Supports CWE-480** - *Use of Incorrect Operator*

**Supports CWE-481** - *Assigning instead of Comparing*

**Supports CWE-783** - *Operator Precedence Logic Error*

### 721 if statement has empty body

**info** A semi-colon was found immediately to the right of a right parenthesis in a construct of the form `if(e);`. As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the `;` is separated by at least one blank from the `)`. Better, place it on a separate line. See also message [548](#).

**Supports CERT C EXP15-C** - *Do not place a semicolon on the same line as an if, for, or while statement*

**Supports CWE-398** - *Indicator of Poor Code Quality*

### 722 'context' statement has empty body

**info** A semi-colon was found immediately to the right of a right parenthesis in a construct of the form `while(e);` or `for(e; e; e);`. As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the `;` is separated by at least one blank from the `)`. Better, place it on a separate line.

**Supports CERT C EXP15-C** - *Do not place a semicolon on the same line as an if, for, or while statement*

**Supports CWE-398** - *Indicator of Poor Code Quality*



**723 macro definition starting with = is suspicious**

**info** A preprocessor definition began with an = sign. For example:

```
#define LIMIT = 50
```

Was this intentional? Or was the programmer thinking of assignment when he wrote this?

**725 unexpected lack of indentation**

**info** The current line was found to be aligned with, rather than indented with respect to, the indicated line. The indicated line corresponds to a clause introducing a control structure and statements within its scope are expected to be indented with respect to it. If tabs within your program are other than 8 blanks you should use the `-t` option (See Section [17.4 Indentation Checking](#)).

**Supports** **CWE-483** - *Incorrect Block Delimitation*

**726 extraneous comma ignored at end of enumerator list after enumerator *symbol***

**info** A comma followed by a right-brace within an enumeration is not a valid ANSI/ISO construct. The comma is ignored. This message is only emitted in C90 and C++03 modes as later versions allow this construct.

**727 static local symbol *symbol* not explicitly initialized**

**info** The named static variable (local to a function) was not explicitly initialized before use. The following remarks apply to messages [728](#) and [729](#) as well as 727. By no explicit initialization we mean that there was no initializer present in the definition of the object, no direct assignment to the object (or any of its elements or members), and no address operator applied to the object or, if the address of the object was taken, it was assigned to a pointer to const. Arrays are also considered to be explicitly initialized if the result of array to pointer decay is assigned to a non-const pointer.

These messages do not necessarily signal errors since the implicit initialization for static variables is 0. However, the messages are helpful in indicating those variables that you had forgotten to initialize to a value. To extract the maximum benefit from the messages we suggest that you employ an explicit initializer for those variables that you want to initialize to 0. For example:

```
static int n = 0;
```

For variables that will be initialized dynamically, do not use an explicit initializer as in:

```
static int m;
```

This message will be given for any array, `class`, `struct` or `union` if no member or element has been assigned a value. The `fde` flag controls how initialization using a default constructor is interpreted in C++.

**728 file scope static variable *symbol* not explicitly initialized**

**info** The named intra-module variable (static variable with file scope) was not explicitly initialized. See the comments on message [727](#) for more details.

**729 external variable *symbol* not explicitly initialized**

**info** The named inter-module variable (external variable) was not explicitly initialized. See the comments on message [727](#) for more details. This message is not issued for library symbols and suppressed for unit checkout (`-unit_check`) and will not report C++ classes.

**730 boolean used as argument *integer to function symbol***

**info** A Boolean was used as an argument to a function. Was this intended? Or was the programmer confused by a particularly complex conditional statement? Experienced C programmers often suppress this message. This message is given only if the associated parameter is not declared bool.

**731 boolean argument(s) to *equality-operator***

**info** A Boolean operator was used as an argument to == or !=. For example:

```
if( (a > b) == (c > d) )      ...
```

tests to see if the inequalities are of the same value. This could be an error as it is an unusual use of a Boolean (see Warnings 503 and 514) but it may also be deliberate since this is the only way to efficiently achieve equivalence or exclusive or. Because of this possible use, the construct is given a relatively mild 'informational' classification. If the Boolean argument is cast to some type, this message is not given. Additionally, this message is not necessarily given just because one of the arguments to == or != is a Boolean type but only if at least one of the arguments is expressed using a Boolean operator. For example, if `e` and `f` are of type bool, the clause:

```
if( e == f ) ...
```

will not prompt this message. However,

```
if( e == !f ) ...
```

will.

**Supports CERT C EXP13-C - Treat relational and equality operators as if they were nonassociative**

**732 loss of sign (*context*) (*type to type*)**

**info** An assignment (or implied assignment, see *context*) is made from a signed quantity to an unsigned quantity. Also, it could not be determined that the signed quantity had no sign. For example:

```
u = n;      /* Info 732 */
u = 4;      /* OK      */
```

where `u` is unsigned and `n` is not, warrants a message only for the first assignment, even though the constant 4 is nominally a signed `int`.

Make sure that this is not an error (that the assigned value is never negative) and then use a cast (to unsigned) to remove the message.

**Supports CERT C INT02-C - Understand integer conversion rules**

**Supports CWE-192 - Integer Coercion Error**

**Supports CWE-195 - Signed to Unsigned Conversion Error**

**Supports CWE-197 - Numeric Truncation Error**

**Supports CWE-562 - Return of Stack Variable Address**

**Supports CWE-681 - Incorrect Conversion between Numeric Types**

**Supports CWE-704 - Incorrect Type Conversion or Cast**

**733 likely assigning address of local *symbol* to outer scope pointer *symbol***

**info** The address of an `auto` variable is valid only within the block in which the variable is declared. An address to such a variable has been assigned to a variable that has a longer life expectancy. There is an inherent danger in doing this.

**Supports CERT C DCL30-C - Declare objects with appropriate storage durations**

**Supports MISRA C 2012 Rule 18.6**

**Supports MISRA C 2004 Rule 17.6**

**Supports CWE-562 - Return of Stack Variable Address**

**734 loss of precision (*context*) from *number* bits to *number* bits**

**info** An assignment is being made into an object smaller than an `int`. The information being assigned is derived from another object or combination of objects in such a way that information could potentially be lost. The number of bits given does not count the sign bit. For example if `ch` is a `char` and `n` is an `int` then:

```
ch = n;
```

will trigger this message whereas:

```
ch = n & 1;
```

will not. To suppress the message a cast can be made as in:

```
ch = (char) n;
```

You may receive notices involving multiplication and shift operators with subinteger variables. For example:

```
ch = ch << 2;
ch = ch * ch;
```

where, for example, `ch` is an `unsigned char`. These can be suppressed by using the flag `+fpm` (precision of an operator is bound by the maximum of its operands).

**Supports CERT C INT02-C - Understand integer conversion rules**

**Supports CWE-192 - Integer Coercion Error**

**Supports CWE-197 - Numeric Truncation Error**

**735 implicit conversion (*context*) from *type* to *type***

**info** An assignment (or implied assignment, see *context*) is made from a `long double` to a `double`. Using a cast will suppress the message. The number of bits includes the sign bit.

**Supports CERT C FLP34-C - Ensure that floating-point conversions are within range of the new type**

**Supports CWE-197 - Numeric Truncation Error**

**Supports CWE-681 - Incorrect Conversion between Numeric Types**

**736 loss of precision (*context*) from *number* bits to *number* bits**

**info** An assignment (or implied assignment, see *context*) is being made to a `float` from a value or combination of values that appear to have higher precision than a `float`. You may suppress this message by using a cast. The number of bits includes the sign bit.

**Supports CERT C FLP03-C - Detect and handle floating-point errors**

**Supports CERT C FLP34-C - Ensure that floating-point conversions are within range of the new type**

**Supports CWE-197 - Numeric Truncation Error**

**Supports CWE-369 - Divide By Zero**

**Supports CWE-681 - Incorrect Conversion between Numeric Types**

**737 loss of sign in promotion from *type* to *type***

**info** An unsigned quantity was joined with a signed quantity in a binary operator (or 2nd and 3rd arguments to the conditional operator) and the signed quantity is implicitly converted to `unsigned`. The message will not be given if the signed quantity is an unsigned constant, a Boolean, or an expression involving bit manipulation. For example,

```
u & ~0xFF
```

where `u` is unsigned does not draw the message even though the operand on the right is technically a signed integer constant. It looks enough like an unsigned to warrant not giving the message.

This mixed mode operation could also draw Warnings [573](#) or [574](#) depending upon the operator involved.

You may suppress the message with a cast but you should first determine whether the signed value could ever be negative or whether the unsigned value can fit within the constraints of a signed quantity.

**Supports CERT C INT02-C** - *Understand integer conversion rules*

**Supports CERT C API09-C** - *Compatible values should have the same type*

**Supports CWE-192** - *Integer Coercion Error*

**Supports CWE-195** - *Signed to Unsigned Conversion Error*

**Supports CWE-197** - *Numeric Truncation Error*

### **738 address of static local symbol *symbol* not explicitly initialized before passed to a function**

**info** The named static local variable was not initialized before being passed to a function whose corresponding parameter is declared as pointer to `const`. Is this an error or is the programmer relying on the default initialization of 0 for all static items? By employing an explicit initializer you will suppress this message. See also message numbers [727](#) and [603](#).

**Supports CWE-456** - *Missing Initialization of a Variable*

### **739 trigraph sequence '*string*' in string literal**

**info** The indicated Trigraph (three-character) sequence was found within a string. This trigraph reduces to a single character according to the ANSI/ISO standards. This represents a "Quiet Change" from the past where the sequence was not treated as exceptional. If you had no intention of mapping these characters into a single character you may precede the initial '?' with a backslash. If you are aware of the convention and you intend that the Trigraph be converted you should suppress this informational message.

**Supports AUTOSAR17 Rule A2-5-1**

**Supports AUTOSAR19 Rule A2-5-1**

**Supports MISRA C 2012 Rule 4.2**

**Supports MISRA C++ Rule 2-3-1**

**Supports MISRA C 2004 Rule 4.2**

### **742 multi-character character constant**

**info** A character constant was found that contained multiple characters, e.g., `'ab'`. This is legal C but the numeric value of the constant is implementation defined. It may be safe to suppress this message because, if more characters are provided than what can fit in an `int`, message number [25](#) is given.

### **743 negative character constant**

**info** A character constant was specified whose value is some negative integer. For example, on machines where a byte is 8 bits, the character constant `'\xFF'` is flagged because its value (according to the ANSI/ISO standard) is -1 (its type is `int`). Note that its value is not `0xFF`.

### **744 switch statement has no default**

**info** A `switch` statement has no section labeled `default`. Was this an oversight? It is standard practice in many programming groups to always have a `default: case`. This can lead to better (and earlier) error detection. One way to suppress this message is by introducing a vacuous `default break;` statement. If you think this adds too much overhead to your program, think again. In all cases tested so far, the introduction of this statement added absolutely nothing to the overall length of code. If you accompany the vacuous statement with a suitable comment, your code will at least be more readable.

This message is not given if the control expression is an enumerated type. In this case, all enumerated constants are expected to be represented by `case` statements, else [787](#) will be issued.

Supports AUTOSAR17 Rule M6-4-6

Supports AUTOSAR19 Rule M6-4-6

Supports CERT C MSC01-C - *Strive for logical completeness*

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.4

Supports MISRA C++ Rule 6-4-6

Supports CWE-478 - *Missing Default Case in Multiple Condition Expression*

#### 746 call to function *symbol* not made in the presence of a prototype

**info** A call to a function is not made in the presence of a prototype. This does not mean that PC-lint Plus is unaware of any prototype; it means that a prototype is not in a position where a compiler will see it. If you have not adopted a strict prototyping convention you will want to suppress this message with `-e746`.

Supports CERT C DCL07-C - *Include the appropriate type information in function declarators*

Supports CERT C DCL31-C - *Declare identifiers before using them*

Supports MISRA C 2004 Rule 8.1

Supports CWE-685 - *Function Call With Incorrect Number of Arguments*

#### 749 local enumeration constant *symbol* not referenced

**info** A member (name provided as *symbol*) of an `enum` was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. Compare with messages [754](#) and [769](#).

Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

Supports CWE-561 - *Dead Code*

#### 750 local macro '*string*' not referenced

**info** A 'local' macro is one that is not defined in a header file. The macro is not referenced throughout the module in which it is defined.

Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

Supports MISRA C 2012 Rule 2.5

Supports CWE-561 - *Dead Code*

#### 751 local typedef *symbol* not referenced

**info** A 'local' typedef symbol is one that is not defined in any header file. It may have file scope or block scope but it was not used through its scope.

Supports AUTOSAR17 Rule M0-1-5

Supports AUTOSAR19 Rule A0-1-6

Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

Supports MISRA C 2012 Rule 2.3

Supports MISRA C++ Rule 0-1-5

Supports CWE-561 - *Dead Code*

#### 752 local declarator *symbol* not referenced

**info** A 'local' declarator symbol is one declared in a declaration appearing in the module file itself as opposed to a header file. The symbol may have file scope or may have block scope. But it wasn't referenced.

Supports AUTOSAR17 Rule M0-1-3

Supports AUTOSAR19 Rule M0-1-3

Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*

Supports MISRA C++ Rule 0-1-3

Supports CWE-561 - *Dead Code*

**753 local *string* symbol not referenced**

**info** *string* is one of **struct**, **class**, **union**, or **enum** and *symbol* is the name of the tag. A 'local' tag is one not defined in a header file. Since its definition appeared, why was it not used? Use of a tag is implied by the use of any of its members.

**Supports AUTOSAR17 Rule M0-1-5**

**Supports AUTOSAR19 Rule A0-1-6**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C 2012 Rule 2.4**

**Supports MISRA C++ Rule 0-1-5**

**Supports CWE-561 - Dead Code**

**754 local *string* member *symbol* not referenced**

**info** A member (name provided as *symbol*) of a **struct**, **class**, or **union** (as indicated in *string*) was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. See message [768](#).

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C 2012 AMD4 Rule 2.8**

**Supports CWE-561 - Dead Code**

**755 global macro '*string*' not referenced**

**info** A 'global' macro is one defined in a header file. The macro is not used in any of the modules comprising the program. This message is suppressed for unit checkout ([-unit\\_check](#) option).

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C 2012 Rule 2.5**

**Supports CWE-561 - Dead Code**

**756 global typedef *symbol* not referenced**

**info** This message is given for a **typedef** symbol declared in a non-library header file. The symbol is not used in any of the modules comprising a program. This message is suppressed for unit checkout ([-unit\\_check](#) option).

**Supports AUTOSAR17 Rule M0-1-5**

**Supports AUTOSAR19 Rule A0-1-6**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C 2012 Rule 2.3**

**Supports MISRA C++ Rule 0-1-5**

**Supports CWE-561 - Dead Code**

**757 global declarator *symbol* not referenced**

**info** A 'global' declarator is one defined in a header file. This message is given for objects declared in non-library header files that have not been used in any module comprising the program being checked. The message is suppressed for unit checkout ([-unit\\_check](#)).

**Supports AUTOSAR17 Rule M0-1-3**

**Supports AUTOSAR19 Rule M0-1-3**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C++ Rule 0-1-3**

**Supports CWE-561 - Dead Code**

**758 global *string* symbol not referenced**

**info**

A 'global' tag is one that is defined in a header file. This message is given for **struct**, **union** and **enum** tags that have been defined in non-library header files and that have not been used in any module comprising the program. The message is suppressed for unit checkout (**-unit\_check**).

**Supports AUTOSAR17 Rule M0-1-5**

**Supports AUTOSAR19 Rule A0-1-6**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports MISRA C++ Rule 0-1-5**

**Supports CWE-561 - Dead Code**

#### **759 header declaration for symbol *symbol* could be moved from header to module**

**info** This message is given for declarations of non-library symbols that are not referenced outside the defining module. Hence, it can be moved inside the module and thereby 'lighten the load' on all modules using the header. This message is suppressed for unit checkout (**-unit\_check** option).

**Supports MISRA C++ Rule 3-3-1**

#### **760 redundant macro *name* defined identically**

**info** The given macro was defined earlier in the same way and is hence redundant.

**Supports MISRA C 2012 Rule 5.4**

#### **761 redundant typedef *name***

**info** A typedef symbol has been declared earlier and is redundant. Although the declarations are consistent you should probably remove the second.

#### **764 switch with no cases**

**info** A **switch** statement has been found that does not have a **case** statement associated with it (it may or may not have a **default** statement). This is normally a useless construct.

**Supports MISRA C 2012 Rule 16.1**

**Supports MISRA C 2012 Rule 16.6**

**Supports MISRA C++ Rule 6-4-8**

**Supports MISRA C 2004 Rule 15.5**

#### **765 external symbol *symbol* could be made static**

**info** An external symbol was referenced in only one module. It was not declared **static**. Some programmers like to make **static** every symbol they can, because this lightens the load on the linker. It also represents good documentation. This message is not issued for library symbols and is suppressed for unit checkout (**-unit\_check** option).

**Supports CERT C DCL15-C - Declare file-scope objects or functions that do not need external linkage as static**

**Supports CERT C DCL19-C - Minimize the scope of variables and functions**

**Supports MISRA C 2012 Rule 8.7**

**Supports MISRA C++ Rule 3-3-1**

**Supports MISRA C 2004 Rule 8.10**

#### **767 macro '*string*' was defined differently in another module**

**info** Two macros processed in two different modules had inconsistent definitions.

#### **768 global structure member *symbol* not referenced**

**info** A member (name provided as *symbol*) of a **struct** or **union** appeared in a non-library header file but was not



used in any module comprising the program. This message is suppressed for unit checkout. Since a **struct** may be replicated in storage, finding an unused member can pay handsome storage dividends. However, many structures merely reflect an agreed upon convention for accessing storage and for any one program, many members are unused. In this case, receiving this message can be a nuisance. One convenient way to avoid unwanted messages (other than the usual **-e** and **-esym**) is to always place such structures in library header files. Alternatively, you can place the struct within a **++flb ... --flb** sandwich to force it to be considered library.

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports MISRA C 2012 AMD4 Rule 2.8**

**Supports CWE-561** - *Dead Code*

#### 769 global enumeration constant *symbol* not referenced

**info** A member (name provided as *symbol*) of an **enum** appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. There are reasons why a programmer may occasionally want to retain an unused **enum** and for this reason this message is distinguished from 768 (unused member). See message 768 for ways of selectively suppressing this message.

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports CWE-561** - *Dead Code*

#### 770 tag *symbol* defined identically at *location*

**info** The **struct**, **union**, or **enum** tag *symbol* was defined identically in different locations (usually two different files). This is not an error but it is not necessarily good programming practice either. It is better to place common definitions of this kind in a header file where they can be shared among several modules. If you do this, you will not get this message. Note that if the tag is defined differently in different scopes, you will receive warning 631 rather than this message. This message is suppressed for unit checkout (**-unit\_check** option).

#### 773 expression-like macro '*string*' not parenthesized

**info** A macro that appeared to be an expression contained unparenthesized binary operators and therefore may result in unexpected associations when used with other operators. For example,

```
#define A B + 1
```

may be used later in the context:

```
f( A * 2 );
```

with the surprising result that B+2 gets passed to **f** and not the (B+1)\*2. Corrective action is to define A as:

```
#define A (B + 1)
```

Highest precedence binary operators are not reported upon. Thus:

```
#define A s.x
```

does not elicit this message because this case does not seem to represent a problem. Also, unparenthesized unary operators (including casts) do not generate this message. [8, Section 19.5]

**Supports CERT C PRE02-C** - *Macro replacement lists should be parenthesized*

#### 774 boolean condition for '*detail*' always evaluates to '*detail*'

**info** The indicated clause (*detail* is one of **if**, **while** or **for** (2nd expression)) has an argument that appears to always evaluate to either '**true**' or '**false**' (as indicated in the message). Information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message 506, which is based on testing constants or combinations of constants. Also compare with the Elective Note 944, which can sometimes provide more detailed information. See Chapter 8 [Value Tracking](#).



Supports AUTOSAR17 Rule M0-1-1  
 Supports AUTOSAR17 Rule M0-1-2  
 Supports AUTOSAR17 Rule M0-1-9  
 Supports AUTOSAR19 Rule M0-1-1  
 Supports AUTOSAR19 Rule M0-1-2  
 Supports AUTOSAR19 Rule M0-1-9  
 Supports CERT C MSC12-C - *Detect and remove code that has no effect or is never executed*  
 Supports MISRA C 2012 Rule 2.2  
 Supports MISRA C 2012 Rule 14.3  
 Supports MISRA C++ Rule 0-1-1  
 Supports MISRA C++ Rule 0-1-2  
 Supports MISRA C++ Rule 0-1-9  
 Supports MISRA C 2004 Rule 13.7  
 Supports CWE-561 - *Dead Code*  
 Supports CWE-570 - *Expression is Always False*  
 Supports CWE-571 - *Expression is Always True*

#### 775 nonnegative quantity cannot be less than zero

info

A non-negative quantity is being compared for being  $\leq 0$ . This is a little suspicious since a non-negative quantity can be equal to 0 but never less than 0. The non-negative quantity may be of type `unsigned` or may have been promoted from an `unsigned` type or may have been judged not to have a sign by virtue of it having been AND'ed with a quantity known not to have a sign bit, an `enum` that may not be negative, etc. See also Warning [568](#).

Supports CWE-398 - *Indicator of Poor Code Quality*

#### 776 possible truncation of addition

info

An `int` expression (signed or unsigned) involving addition or subtraction is converted to `long` implicitly or explicitly. Moreover, the precision of a `long` is greater than that of `int`. If an overflow occurred, information would be lost. Either cast one of the operands to some form of `long` or cast the result to some form of `int`.

See Warning [647](#) for a further description and an example of this kind of error. See also messages [790](#) and [942](#).

Supports CERT C INT08-C - *Verify that all integer values are in range*

Supports CWE-681 - *Incorrect Conversion between Numeric Types*

#### 777 testing floating point values for equality

info

This message is issued when the operands of operators `==` and `!=` are some form of floating type (`float`, `double`, or `long double`). Testing for equality between two floating point quantities is suspect because of round-off error and the lack of perfect representation of fractions. If your numerical algorithm calls for such testing turn the message off. The message is suppressed when one of the operands can be represented exactly, such as 0 or 13.5.

Supports AUTOSAR17 Rule M6-2-2

Supports AUTOSAR19 Rule M6-2-2

Supports CERT C FLP00-C - *Understand the limitations of floating-point numbers*

Supports CERT C FLP02-C - *Avoid using floating-point numbers when precise computation is needed*

Supports MISRA C++ Rule 6-2-2

Supports MISRA C 2004 Rule 13.3

Supports CWE-398 - *Indicator of Poor Code Quality*

#### 778 constant expression evaluates to 0 in 'unary/binary' operation 'operator'

info

A constant expression involving addition, subtraction, multiplication, shifting, or negation resulted in a 0. This could be a purposeful computation but could also have been unintended. If this is intentional, suppress the message. If one of the operands is 0, Elective Note [941](#) may be issued rather than a [778](#).

**779 string constant in comparison operator '*operator*'**

**info** A string constant appeared as an argument to a comparison operator. For example:

```
if( s == "abc" ) ...
```

This is usually an error. Did the programmer intend to use `strcmp`? It certainly looks suspicious. At the very least, any such comparison is bound to be machine-dependent. If you cast the string constant, the message is suppressed.

**Supports CWE-597** - *Use of Wrong Operator in String Comparison*

**783 line does not end with a newline**

**info** This message is issued when an input line is not terminated by a new-line or when a NUL character appears within an input line. If your editor is in the habit of not appending new-lines onto the end of the last line of the file then suppress this message. Otherwise, examine the file for NUL characters and eliminate them.

**784 nul character truncated from string**

**info** During initialization of an array with a string constant there was not enough room to hold the trailing NUL character. For example:

```
char a[3] = "abc";
```

would evoke such a message. This may not be an error since the easiest way to do this initialization is in the manner indicated. It is more convenient than:

```
char a[3] = { 'a', 'b', 'c' };
```

On the other hand, if it really is an error it may be especially difficult to find.

**Supports CERT C STR11-C** - *Do not specify the bound of a character array initialized with a string literal*

**Supports CWE-170** - *Improper Null Termination*

**Supports CWE-464** - *Addition of Data Structure Sentinel*

**785 too few initializers for aggregate of type *type* in initialization of *symbol***

**info** The number of initializers in a brace-enclosed initializer was less than the number of items in the aggregate. Default initialization is taken. An exception is made with the initializer `{0}`. This is given a separate message number in the Elective Note category ([943](#)). It is normally considered to be simply a stylized way of initializing all members to 0. See also [9068](#).

**786 string concatenation within initializer**

**info** Although it is perfectly 'legal' to concatenate string constants within an initializer, this is a frequent source of error. Consider:

```
char *s[] = { "abc" "def" };
```

Did the programmer intend to have an array of two strings but forget the comma separator? Or was a single string intended?

**787 enum constant *symbol* not used within switch**

**info** A `switch` expression is an enumerated type and at least one of the enumerated constants was not present as

a `case` label. Moreover, no `default` case was provided.

Supports AUTOSAR17 Rule M6-4-6

Supports AUTOSAR19 Rule M6-4-6

Supports CERT C MSC01-C - *Strive for logical completeness*

Supports MISRA C++ Rule 6-4-6

### 788 enum constant *symbol* not used within default switch

**info** A `switch` expression is an enumerated type and at least one of the enumerated constants was not present as a `case` label. However, unlike Info 787, a `default` case was provided. This is a mild form of the case reported by Info 787. The user may thus elect to inhibit this mild form while retaining Info 787.

### 789 assigning address of auto variable *symbol* to static

**info** The address of an `auto` variable (*symbol*) is being assigned to a `static` variable. This is dangerous because the `static` variable will persist after return from the function in which the `auto` is declared but the `auto` will be, in theory, gone. This can prove to be among the hardest bugs to find. If you have one of these, make certain there is no error and use `-esym` to suppress the message for a particular variable.

Supports AUTOSAR17 Rule M7-5-2

Supports AUTOSAR19 Rule M7-5-2

Supports CERT C DCL30-C - *Declare objects with appropriate storage durations*

Supports MISRA C 2012 Rule 18.6

Supports MISRA C++ Rule 7-5-2

Supports MISRA C 2004 Rule 17.6

Supports CWE-562 - *Return of Stack Variable Address*

### 790 possibly truncated *string* promoted to *type*

**info** This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals, the result of which is later converted to a floating point quantity. The operations that are scrutinized and reported upon by this message are: shift left and multiplication. Addition and subtraction are covered by note 942. See also messages 647 and 776.

Supports CERT C FLP06-C - *Convert integers to floating point for floating-point operations*

Supports CWE-681 - *Incorrect Conversion between Numeric Types*

### 791 a single line suppression followed a normal option

**info** A temporary message suppression option (one having the form: `!e...`) followed a regular option. Was this intended?

### 792 casting void expression to void

**info** A void expression has been cast to void. Was this intended?

### 793 ANSI/ISO minimum translation limit of *integer* '*string*' exceeded, processing is unaffected

**info** An ANSI/ISO minimum translation limit has been exceeded. These limits are described under the heading "Translation limits" in the ANSI/ISO C Standards and under the heading "Implementation Quantities" in the C++ standards. Programs exceeding these limits are not considered maximally portable. However, they may work for individual compilers.

The *integer* parameter indicates the numeric value that was exceeded and *string* provides a textual description of the limit in question.

Say a large program exceeds the ANSI/ISO limit of 4095 external identifiers. This will result in the message:

```
793 ANSI/ISO minimum translation limit of 4095 'external identifiers'
    exceeded, processing is unaffected
```

It may not be obvious how to inhibit this message for identifiers while leaving other limits in a reportable state. The second parameter of the message is designated *string* and so the `-estring` may be used. Because the string contains a blank, quotes must be used. The option becomes:

```
-estring(793,"external identifiers')
```

See [17.10 Language Limits](#) for additional information and a list of supported limits.

**Supports CERT C DCL40-C** - *Do not create incompatible declarations of the same function or object*

**Supports MISRA C 2012 Rule 1.1**

#### 798 redundant char '*character*'

**info** The indicated character is redundant and can be eliminated from the input source. A typical example is a backslash on a line by itself.

#### 799 numerical constant '*integer*' larger than unsigned long

**info** An integral constant was found to be larger than the largest value allowed for `unsigned long` quantities. By default, an `unsigned long` is 4 bytes but can be respecified via the option `-sl#`. If the `long long` type is permitted (see option `+f11`) this message is automatically suppressed. See also message [417](#).

#### 801 goto statement used

**info** A `goto` was detected. Use of the `goto` is not considered good programming practice by most authors and its use is normally discouraged. There are a few cases where the `goto` can be effectively employed but often these can be rewritten just as effectively without the `goto`. The use of `goto` statements can have a devastating effect on the structure of large functions creating a mass of spaghetti-like confusion. For this reason its use has been banned in many venues.

**Supports AUTOSAR17 Rule A6-6-1**

**Supports AUTOSAR19 Rule A6-6-1**

**Supports MISRA C 2012 Rule 15.1**

**Supports MISRA C 2004 Rule 14.4**

#### 805 expected L"..." to initialize wide char string

**info** An initializer for a wide character array or pointer did not use a preceding 'L'. For example:

```
wchar_t a[] = "abc";
```

was found whereas

```
wchar_t a[] = L"abc";
```

was expected.

#### 806 small signed bitfield

**info** A small bit field (less than an `int` wide) was found and the base type is signed rather than unsigned. Since the most significant bit is a sign bit, this practice can produce surprising results. For example,

```
struct { int b:1; } s;
s.b = 1;
if( s.b > 0 ) /* should succeed but actually fails */
...

```

**808 no explicit type given, int assumed**

**info** An explicit type was missing in a declaration. Unlike Warning [601](#), the declaration may have been accompanied by a storage class or modifier (qualifier) or both. For example:

```
extern f(void);
```

will draw message 808. Had the **extern** not been present, a **745** would have been raised.

The keywords **unsigned**, **signed**, **short** and **long** are taken to be explicit type specifiers even though **int** is implicitly assumed as a base.

**Supports CERT C DCL31-C - Declare identifiers before using them**

**Supports MISRA C 2012 Rule 8.1**

**Supports MISRA C 2004 Rule 8.2**

**809 likely returning address of local *symbol* through *symbol***

**info** The value held by a pointer variable may have been the address of an **auto** variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

**810 arithmetic modification of custodial pointer**

**info** We define the custodial variable as that variable directly receiving the result of a **malloc** or **new** or equivalent call. It is inappropriate to modify such a variable because it must ultimately be **free**'ed or **delete**'ed. You should first make a copy of the custodial pointer and then modify the copy. The copy is known as an alias.

**812 static variable *symbol* is *integer* bytes**

**info** The amount of storage for a static symbol has reached or exceeded a value that was specified in a **-size** option.

**813 auto variable *symbol* is *integer* bytes**

**info** The amount of storage for an auto symbol has reached or exceeded a value that was specified in a **-size** option.

**814 tagless struct without a declarator is useless here**

**info** A tagless struct was declared without a declarator. For example:

```
struct { int n; };
```

Such a declaration cannot very well be used.

**815 unsaved pointer used in pointer arithmetic**

**info** An allocation expression (**malloc**, **calloc**, **new**) is not immediately assigned to a variable but is used as an operand in some expression. This would make it difficult to free the allocated storage. For example:

```
p = new X[n] + 2;
```

will elicit this message. A preferred sequence is:

```
q = new X[n];
p = q+2;
```

In this way the storage may be freed via the custodial pointer `q`.

Another example of a statement that will yield this message is:

```
p = new (char *) [n];
```

This is a gruesome blunder on the part of the programmer. It does NOT allocate an array of pointers as a novice might think. It is parsed as:

```
p = (new (char *)) [n];
```

which represents an allocation of a single pointer followed by an index into this 'array' of one pointer.

#### 816 non-ISO format specification '*string*'

**info** A non-standard format specifier was found in a format-processing function such as `printf` or `scanf`. The format was recognized as being a common extension. If the format was not recognized, a more severe warning (557) would have been issued. The non-ISO conversion specifiers that are recognized are:

|                 |                                           |                                                                |
|-----------------|-------------------------------------------|----------------------------------------------------------------|
| <code>%C</code> | <code>wchar_t</code>                      | XSI/MS extension, equivalent to <code>%lc</code>               |
| <code>%D</code> | <code>int</code>                          | Apple extension, synonym for <code>%d</code>                   |
| <code>%O</code> | <code>unsigned int</code>                 | Apple extension, synonym for <code>%o</code>                   |
| <code>%S</code> | <code>wchar_t *</code>                    | XSI/MS extension, equivalent to <code>%ls</code>               |
| <code>%U</code> | <code>unsigned int</code>                 | Apple extension, synonym for <code>%u</code>                   |
| <code>%Z</code> | <code>ANSI_STRING / UNICODE_STRING</code> | MS extension                                                   |
| <code>%m</code> | <code>none</code>                         | Glibc extension, prints output of <code>strerror(errno)</code> |

**Supports CERT C FIO47-C** - Use valid format strings

**Supports CWE-134** - Use of Externally-Controlled Format String

**Supports CWE-685** - Function Call With Incorrect Number of Arguments

**Supports CWE-686** - Function Call With Incorrect Argument Type

#### 818 parameter *symbol* of function *symbol* could be pointer to const

**info** As an example:

```
int f( int *p ) { return *p; }
```

can be redeclared as:

```
int f( const int *p ) { return *p; }
```

Declaring a parameter a pointer to `const` offers advantages that a mere pointer does not. In particular, you can pass to such a parameter the address of a `const` data item. In addition it can offer better documentation.

Other situations in which a `const` can be added to a declaration are covered in messages 952, 953, 954 and 1764.

**Supports AUTOSAR17 Rule M7-1-2**

**Supports AUTOSAR19 Rule M7-1-2**

**Supports CERT C DCL13-C** - Declare function parameters that are pointers to values not changed by the function as `const`

**Supports MISRA C 2012 Rule 8.13**

**Supports MISRA C++ Rule 7-1-2**

**Supports MISRA C 2004 Rule 16.7**

#### 820 boolean test of parenthesized assignment

**info** A Boolean test was made on the result of an assignment and, moreover, the assignment was parenthesized. For example:

```
if ( ( a = b ) ) ... // Info 820
```

will draw this informational whereas

```
if ( a = b ) ... // Info 720
```

(i.e. the unparenthesized case) will, instead, draw Info 720. We, of course, do not count the outer parentheses, required by the language, that always accompany the `if` clause.

The reason for partitioning the messages in this fashion is to allow the programmer to adopt the convention, advanced by some compilers (in particular `gcc`), of always placing a redundant set of parentheses around any assignment that is to be tested. In this case you can suppress Info 820 (via `-e820`) while still enabling Info 720.

**Supports AUTOSAR17 Rule M6-2-1**

**Supports AUTOSAR19 Rule M6-2-1**

**Supports MISRA C 2012 Rule 13.4**

**Supports MISRA C++ Rule 6-2-1**

### 821 right hand side of assignment not parenthesized

**info** An assignment operator was found having one of the following forms:

```
a = b || c
a = b && c
a = b ? c : d
```

Moreover, the assignment appeared in a context where a value was being obtained. For example:

```
f( a = b ? c : d );
```

The reader of such code could easily confuse the assignment for a test for equality. To eliminate any such doubts we suggest parenthesizing the right hand side as in:

```
f( a = (b ? c : d) );
```

### 823 definition of macro '*name*' ends in semi-colon

**info** The last token in the replacement text of the specified macro was a semi-colon. In addition to limiting the places where the macro can be used, this can result in unintentional behavior when the macro is used in what appears to be a larger expression. For example, in:

```
#define SUM(x, y) ((x) + (y));
void foo(int a, int b) {
    int result = SUM(a, b) + 1;
}
```

`result` will be the sum of `a` and `b` without the `+ 1`.

**Supports CERT C PRE11-C** - *Do not conclude macro definitions with a semicolon*

### 825 control flow falls through to next case without an intervening -fallthrough comment

**info** A common programming mistake is to forget a `break` statement between case statements of a `switch`. For example:

```
case 'a': a = 0;
case 'b': a++;
```

Is the fall through deliberate or is this a bug? To signal that this is intentional use the `-fallthrough` option within a lint comment as in:

```

    case 'a': a = 0;
    //lint -fallthrough
    case 'b': a++;

```

This message is similar to Warning [616](#) ("control flows into case/default") and is intended to provide a stricter alternative. Warning [616](#) is suppressed by any comment appearing at the point of the fallthrough. Thus, an accidental omission of a break can go undetected by the insertion of a neutral comment. This can be hazardous to well-commented programs.

**Supports CERT C MSC17-C** - *Finish every set of statements associated with a case label with a break statement*

**Supports MISRA C 2012 Rule 16.1**

**Supports MISRA C 2012 Rule 16.3**

**Supports CWE-484** - *Omitted Break Statement in Switch*

## 826 suspicious pointer-to-pointer conversion (area too small)

info

A pointer was converted into another either implicitly or explicitly. The area pointed to by the destination pointer is larger than the area that was designated by the source pointer. For example:

```

long *f( char *p ) { return (long *) p; }

```

**Supports CERT C MEM35-C** - *Allocate sufficient memory for an object*

**Supports CWE-131** - *Incorrect Calculation of Buffer Size*

**Supports CWE-190** - *Integer Overflow or Wraparound*

**Supports CWE-467** - *Use of sizeof() on a Pointer Type*

**Supports CWE-680** - *Integer Overflow to Buffer Overflow*

**Supports CWE-789** - *Memory Allocation with Excessive Size Value*

## 827 loop can only be reached via goto due to unconditional transfer of control by 'string' statement

info

A loop structure (for, while, or do) could not be reached. Was this an oversight? It may be that the body of the loop has a labeled statement and that the plan of the programmer is to jump into the middle of the loop through that label. It is for this reason that we give an Informational message and not the Warning ([527](#)) that we would normally deliver for an unreachable statement. But please note that jumping into a loop is a questionable practice in any regard.

**Supports AUTOSAR17 Rule M0-1-1**

**Supports AUTOSAR17 Rule M0-1-2**

**Supports AUTOSAR19 Rule M0-1-1**

**Supports AUTOSAR19 Rule M0-1-2**

**Supports CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

**Supports MISRA C 2012 Rule 2.1**

**Supports MISRA C++ Rule 0-1-1**

**Supports MISRA C++ Rule 0-1-2**

**Supports MISRA C 2004 Rule 14.1**

**Supports CWE-561** - *Dead Code*

## 828 semantics of 'string' copied to function 'string'

info

A function with built-in semantics or user-defined semantics was `#define`'d to be some other function with a similar name formed by prepending or appending underscores. For example:

```

#define strcmp(a,b) __strcmp__(a,b)

```

will cause Info [828](#) to be issued. As the message indicates, the semantics will be automatically transferred to the new function.



**829 a +headerwarn option was previously issued for header 'file'**

**info** Some coding standards discourage or even prohibit the use of certain header files. PC-lint Plus can guard against their use by activating the lint option `+headerwarn(file)`. Later, if the file is used, we will then issue this message.

Supports AUTOSAR17 Rule A18-0-1

Supports AUTOSAR17 Rule A18-0-3

Supports AUTOSAR19 Rule A18-0-1

Supports AUTOSAR19 Rule A18-0-3

Supports MISRA C 2012 Rule 17.1

Supports MISRA C 2012 Rule 21.4

Supports MISRA C 2012 Rule 21.5

Supports MISRA C 2012 Rule 21.10

Supports MISRA C 2012 Rule 21.11

Supports MISRA C 2012 Rule 21.12

Supports MISRA C++ Rule 18-0-1

Supports MISRA C++ Rule 18-0-4

Supports MISRA C++ Rule 18-7-1

Supports MISRA C++ Rule 27-0-1

Supports MISRA C 2004 Rule 20.8

Supports MISRA C 2004 Rule 20.9

**831 value tracking history *text varies***

**supplemental**

This message provides supplemental value tracking history information and is attached to a proceeding value tracking message. Multiple 831 messages may be provided for single value tracking message and the exact verbiage will vary depending on the situation. For example:

```
short f(short x, short y) {
    if (x >= 10 && x <= 20) {
        return y / (x - 15);
    }
    ....
}
```

results in:

```
warning 414: possible division by zero
    return y / (x - 15);
           ^ ~~~~~~

supplemental 831: operator - yields -5:5
    return y / (x - 15);
           ~~~~

supplemental 831: integral conversion yields 10:20
    return y / (x - 15);
           ^

supplemental 831: inference yields 10:20
    if (x >= 10 && x <= 20) {
    ^ ~~~~~~

supplemental 831: inference yields 10:32767
    if (x >= 10 && x <= 20) {
    ^ ~~~~~~
```

The main message is [414](#) - "possible division by zero" and the 831 messages show the steps that lead PC-lint Plus to make this determination. Starting from the bottom up, the last message indicates the left-hand side of the `&&` operator resulted in an inference that the lower bound of `x` must be 10. The right-hand side served to constrain the upper bound of `x`. At the point of the return statement, PC-lint Plus knows that the value of `x` is between 10 and 20, inclusive. The value of the expression `x - 15` therefore is between -5 and 5. Since this is a constrained range that contains the value zero and is being used as a divisor, message [414](#) is issued.

**832 parameter *symbol* not explicitly declared, int assumed**

**info** In an old-style function definition a parameter was not explicitly declared. To illustrate:

```
void f( n, m )
    int n;
    { ...
```

This is an example of an old-style function definition with `n` and `m` the parameters. `n` is explicitly declared and `m` is allowed to default to `int`. An 832 will be issued for `m`.

**Supports MISRA C 2012 Rule 8.1**

**834 operator '*operator*' followed by operator '*operator*' could be confusing without parentheses**

**info** Some combinations of operators seem to be confusing. For example:

```
a = b - c - d;
a = b - c + d;
a = b / c / d;
a = b / c * d;
```

tend to befuddle the reader. To reduce confusion we recommend using parentheses to make the association of these operators explicit. For example:

```
a = (b - c) - d;
a = (b - c) + d;
a = (b / c) / d;
a = (b / c) * d;
```

in place of the above.

**835 zero given as *string* argument to operator context**

**info** A 0 has been provided as an operand to an arithmetic operator. The name of the operator is provided in the message as well as the side of the operator (`left` or `right`) that had the unusual value. For example:

```
n = n + 0 - m;
```

will produce a message that the right hand operand of operator '+' is zero. In general the operators examined are the binary operators:

```
+ - * / % | & ^ << >>
```

and the unary operators - and +. An enumeration constant whose value is 0 is permitted with operators:

```
+ - >> <<
```

Otherwise a message is issued. For example:

```
enum color { red,
             blue = red+100,      /* ok */
             green = red*0x10    /* 835 */
};
```

The assignment operators that have an arithmetic or bitwise component, such as  `|=` , are also examined. The message given is equivalent to that given with the same operator without the assignment component.

**837 switch condition is a constant expression**

**info** The condition of a `switch` statement is a constant expression as in:

```
switch(5) {
    ...
}
```

While legal, this is suspect since the point of a `switch` statement is usually to specify different actions depending on the value of a variable.

### 838 previous value assigned to *symbol* not used

**info** An assignment statement was encountered that apparently obliterated a previously assigned value that had never had the opportunity of being used. For example, consider the following code fragment:

```
y = 1;
if( n > 0 ) y = 2;
y = 4;           // Informational 838 ...
```

Here we can report that the assignment of 4 to `y` obliterates previously assigned values that were not used. We, of course, cannot report anything unusual about the assignment of 2. This will assign over a prior value of 1 that so far had not been used but the existence of an alternative path means that the value of 1 can still be employed later in the code and is accepted for the time being as reasonable. It is only the final assignment that raises alarm bells. See also Warning message [438](#).

**Supports AUTOSAR17 Rule A0-1-1**

**Supports AUTOSAR17 Rule M0-1-9**

**Supports AUTOSAR19 Rule A0-1-1**

**Supports AUTOSAR19 Rule M0-1-9**

**Supports CERT C MSC12-C - Detect and remove code that has no effect or is never executed**

**Supports CERT C MSC13-C - Detect and remove unused values**

**Supports MISRA C++ Rule 0-1-6**

**Supports MISRA C++ Rule 0-1-9**

**Supports CWE-561 - Dead Code**

**Supports CWE-563 - Assignment to Variable without Use**

### 839 storage class of symbol *symbol* assumed static

**info** A declaration for a symbol that was previously declared `static` in the same module was found without the `'static'` specifier. For example:

```
static void f();
extern void f();    // Info 839
void f() {}         // Info 839
```

By the rules of the language `'static'` wins and the symbol is assumed to have internal linkage. This could be the definition of a previously declared `static` function (as in line 3 of the above example) in which case, by adding the `static` specifier, you will inhibit this message. This could also be a redeclaration of either a function or a variable (as in line 2 of the above example) in which case the redeclaration is redundant.

**Supports AUTOSAR17 Rule M3-3-2**

**Supports AUTOSAR19 Rule M3-3-2**

**Supports CERT C DCL36-C - Do not declare an identifier with conflicting linkage classifications**

**Supports MISRA C 2012 Rule 8.8**

**Supports MISRA C++ Rule 3-3-2**

**Supports MISRA C 2004 Rule 8.11**

### 840 NUL character in string literal

**info** A nul character was found in a string literal. This is legal but suspicious and may have been accidental. This is because a nul character is automatically placed at the end of a string literal and because conventional usage and most of the standard library's string functions ignore information past the first nul character.

**Supports CWE-170** - *Improper Null Termination*

**Supports CWE-398** - *Indicator of Poor Code Quality*

**Supports CWE-463** - *Deletion of Data Structure Sentinel*

**841** **function *symbol* is specified as noreturn and declared with non-void return type *type***

**info** A function was declared with a noreturn specifier but a non-void return type. The noreturn specifier indicates that the function will not return to its caller but a non-void return type implies an opportunity for the function to do so.

**Supports MISRA C 2012 AMD3 Rule 17.10**

**843** **static storage duration variable *symbol* could be made const**

**info** A variable of static storage duration is initialized but never modified thereafter. Was this an oversight? If the intent of the programmer is to not modify the variable, it could and should be declared as `const` [9, Item 3]. See also message [844](#).

**Supports AUTOSAR17 Rule A7-1-1**

**Supports AUTOSAR19 Rule A7-1-1**

**Supports MISRA C++ Rule 7-1-1**

**844** **static storage duration variable *symbol* could be made pointer to const**

**info** The data pointed to by a pointer of static storage duration is never changed (at least not through that pointer). It therefore would be better if the variable were typed pointer to const [9, Item 3]. See also message [843](#).

**Supports MISRA C 2012 Rule 8.13**

**845** **the *left/right* operand to *operator* always evaluates to 0**

**info** An operand that can be deduced to always be 0 has been presented to an arithmetic operator in a context that arouses suspicion. The name of the operator is provided in the message as well as the side of the operator (left or right) that had the unusual value. For example:

```
n = 0;
k = m & n;
```

will produce a message that the right hand operand of operator '`&`' is certain to be zero.

The operands examined are the right hand sides of operators

```
+ - | ||
```

the left hand sides of operators

```
/ %
```

and both sides of operators

```
* & << >> &&
```

The reason that the left hand side of operator `+` (and friends) is not examined for zero is that zero is the identity operation for those operators and hence is often used as an initializing value. For example:

```
sum = 0;
for( ... )
    sum = sum + what_ever;    // OK, no message
```

The message is not issued for arithmetic constant zeros. Message [835](#) is issued in that instance.

The message is also suspended when the expression that evaluates to zero contains side-effects. For example:

```
i = 0;
*(buf + i++) = 'A';    /* Okay */
```

Supports MISRA C 2004 Rule 13.7

#### 846 signedness of bitfield *symbol* of type *type* is implementation-defined

**info** A bit-field was defined with a non-Boolean non-enumeral integer type without an explicit **signed** or **unsigned** specifier. E.g.:

```
int a:5;
```

Most bit fields are more useful when they are **unsigned**. If you want to have a **signed** bit field you must explicitly indicate this as follows:

```
signed int a:5;
```

The same also holds for **typedef**'s. For example,

```
typedef int INT;
typedef signed int SINT;
struct {
    INT a:16;    // info 846
    SINT b:16;   // OK
};
```

It is unusual in C or C++ to distinguish between **signed int** and plain **int**; this is one of those rare cases. Note that this message will not be issued within a C++ module using C++14 or later [10, issue 739].

Supports CERT C INT12-C - Do not make assumptions about the type of a plain int bit-field when used in an expression

#### 847 thread unsafe function '*name*' is called without protection in function '*name*' of thread '*name*' at *location*

A function which has been specified as **thread\_unsafe** (identified by the first *name* parameter) is being called, directly or indirectly, by the thread specified in the third *name* parameter via the function specified by the second *name* parameter outside the protection of a mutex. Supplemental messages provide the call path to the **thread\_unsafe** function from the specified thread.

Thread unsafe functions may typically be called safely outside of a thread-protected region provided no other thread is making such a call and it is safe to suppress this message in such cases. Warning [460](#) will report on unprotected calls to thread unsafe functions by multiple threads. The primary purpose of this message is to report calls to functions that have been designated as "Category 3" (see [Thread Unfriendly Functions](#)).

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

#### 849 enumerator *symbol* reuses the constant value '*integer*' previously used by enumerator *symbol*

**info** Two enumerators have the same value. For example:

```
enum colors { red, blue, green = 1 };
```

will elicit this informational message. This is not necessarily an error and you may want to suppress this message for selected enumerators.

#### 850 for statement index variable *symbol* modified in body

**info** A **for** loop with an identifiable loop index variable was programmed in such a way that the loop body also modifies the index variable. For example:

```

void foo(int *a) {
    for (int i = 0; i < 100; i++) {
        a[i++] = 0;
    }
}

```

In general it is better to restrict modifications of `for` loop index variables to the `for` clause if at all possible. If this is not possible, you can prefix the `for` loop with an appropriate lint comment such as:

```
/*lint -e{850} i is modified in the body of the for loop */
```

Supports AUTOSAR17 Rule M6-5-3

Supports AUTOSAR19 Rule M6-5-3

Supports MISRA C++ Rule 6-5-3

Supports MISRA C 2004 Rule 13.6

### 853 entering nested comment

**info** A `/*` sequence was encountered inside of a C-style comment while the `fnc` (nested comments) flag was ON. Since nested comments have been enabled, the next `*/` sequence that is encountered will only terminate the nested comment, not the containing comment. The purpose of this message is to alert you of this fact.

Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*

### 854 trigraph sequence converted to 'string' character

**info** This message is issued when trigraphs are enabled and a trigraph sequence is replaced.

Supports CERT C PRE07-C - *Avoid using repeated question marks*

### 855 positional arguments are a non-ISO extension

**info** Positional arguments are a POSIX extension to C and will not behave as expected on systems that do not support this extension. PC-lint Plus understands and will diagnose misuse specific to positional arguments via messages [493](#), [494](#), [2401](#), and [2404](#).

Supports CERT C FIO47-C - *Use valid format strings*

Supports CWE-134 - *Use of Externally-Controlled Format String*

Supports CWE-685 - *Function Call With Incorrect Number of Arguments*

Supports CWE-686 - *Function Call With Incorrect Argument Type*

### 856 flag 'string' is ignored when flag '-' is present

**info** Within a format string for a `printf` or `scanf` like function, a combination of flags was used in which one of the flags has no effect in the presence of the other. For example:

```
extern int i;
printf("%-0d", i);
```

Will elicit the message:

```
flag '0' is ignored when flag '-' is present
```

because a left-justified field (requested via the `'-'` flag), cannot be padded with zeroes (requested via the `'0'` flag). Such combinations do not result in undefined behavior but likely represent a programming error.

### 857 argument 1 of type *type* is not compatible with argument 2 of type *type* in call to function *symbol*

The first two arguments in a call to `memcmp`, `memmove`, or `memcpy` are not compatible. Using these functions to compare or copy data between different types may have unexpected results.

**Supports MISRA C 2012 AMD1 Rule 21.15****Supports CWE-843 - Access of Resource Using Incompatible Type ('Type Confusion')****865 detail****info** Message 865 is issued as a result of the `-message` option. For example:

```
#ifndef N
//lint -message(Please supply a definition for N)
#endif
```

will issue the message only if N is undefined. See option `-message`.**866 unusual argument to sizeof****info** An expression used as an argument to `sizeof()` counts as "unusual" if it is not a constant, a symbol, a function call, a member access, a subscript operation (with indices of zero or one), or a dereference of the result of a symbol, scoped symbol, array subscript operation, or function call. Also, since unary '+' could legitimately be used to determine the size of a promoted expression, it does not fall under the category of "unusual". Example:

```
char A[10];
unsigned end = sizeof(A - 1);           // 866: was 'sizeof(A) - 1' intended?
size_of_promoted_char = sizeof(+A[0]); // OK, + makes a difference
size_t s1 = sizeof( end+1 );           // 866: use +end to get promoted type
size_t s2 = sizeof( +(end+1) );        // OK, we won't complain
struct B *p;                           // B is some POD.
B b1;

memcpy( p, &b1, sizeof(&b1) );          // 866: sizeof(b1)intended?

size_t s3 = sizeof(A[0]);               // OK, get the size of an element.
size_t s4 = sizeof(A[2]);               // 866: Not incorrect, but unusual ...
```

**868 degenerate switch encountered****info** A degenerate switch was encountered. This is a braceless `switch`. E.g.:

Now why, you might wonder, would one want such a thing. That would be to create a region of code from which you can breakout at any point. E.g.:

```
REGION {
    alpha();
    if( n < 10 ) break;
    beta();
    if( n < 25 ) break;
    gamma();
}
```

If `REGION` is a suitably defined macro then each `break` taken will take you to just below the region. In this simple example there is not that much of an advantage. But when if conditions explode in complexity this is a very nice feature to have.

To obtain this effect you can define `REGION` as

```
#define REGION switch(1) case 1:
```

To automatically suppress this message in this case use:

```
-emacro( 868, REGION )
```

**870** no '-max\_threads=N' option was encountered prior to the first module; only a single thread  
**info** will be used by default

This message is issued at the end of processing if multiple modules were processed but no `-max_threads` option was used to specify how many concurrent linting threads to employ. By default, PC-lint Plus processes all modules using a single thread. If your hardware has multiple cores or processors, you may be able to substantially speed up the processing time by employing multiple threads using the `-max_threads` option. If a single thread is explicitly requested using `-max_threads=1`, this message will be suppressed.

**879** semantic monikers are '*string*' and '*string*'

**info** This message is emitted for function calls encountered while the `fsf` flag is enabled. It lists the different ways that the function that was called can be specified within a `-sem`, `-printf`, or `-scanf` option. Overloaded functions and function templates can have multiple ways of being specified in these options. For example, given a function with multiple overloads, it is possible to specify a semantic that applies to all overloads or just one, similarly for function templates. See also [9.2.2.4 Overload-Specific Semantics](#)

**882** sizeof applied to parameter *symbol* of function *symbol* declared an incomplete array type *type*

**info** The `sizeof` operator was used with a pointer parameter that was declared using array syntax without a size. Using `sizeof` in this way will yield the size of the pointer, not the size of the array. If an array size is provided, message [682](#) is issued instead.

**Supports CERT C ARR01-C** - Do not apply the sizeof operator to a pointer when taking the size of an array

**Supports MISRA C 2012 AMD1 Rule 12.5**

**Supports CWE-467** - Use of sizeof() on a Pointer Type

**886** preprocessor directive '*name*' encountered in conditionally excluded region is deprecated. *string*

**info** A deprecated preprocessor directive (see the `-deprecate` option) was encountered in a *conditionally excluded region*. For example:

```
//lint -deprecate(ppw, pragma)
#if 0
#pragma BLAH
#endif
```

will cause this message to be issued for the `#pragma` directive.

**Supports AUTOSAR17 Rule A16-0-1**

**Supports AUTOSAR17 Rule A16-6-1**

**Supports AUTOSAR17 Rule A16-7-1**

**Supports AUTOSAR19 Rule A16-0-1**

**Supports AUTOSAR19 Rule A16-6-1**

**Supports AUTOSAR19 Rule A16-7-1**

**888** custom metric limit violated

**info** A metric check created using the `+metric` option was violated.

**891** reference information *text varies*

**supplemental**

This supplemental message is used to convey additional information for a previous message at a different location. For example, this message may be used to reference an earlier declaration, a conflicting definition, etc.



- 892** **did you mean to *multiply/divide* by a factor of type '*strong-type*'?**  
**supplemental** Provides supplemental information about a Strong Type mismatch when it appears that a forgotten conversion factor may be responsible for a dimensional type difference.
- 893** **expanded from macro**  
**supplemental** This supplemental message is given when a message is issued with a location that was the result of a macro expansion. It specifies the macro from which the expansion occurred.
- 894** **during specific walk *detail***  
**supplemental** This supplemental message is issued when a value tracking message is issued during a specific walk and provides additional information about the walk. The location of the call, name of the called function, and the arguments passed will be displayed. This information is rendered as described in section 8.5.
- 896** **semantic expression expands to '*string*'**  
**supplemental** This supplemental message is issued when there is an error processing a semantic that contains a macro expansion. It provides information about the macro that was expanded.
- 897** **in instantiation of *string symbol* triggered here**  
**supplemental** This supplemental message is issued when a message is given within a template instantiation. It provides details of the relevant instantiation.
- 900** **execution completed producing *integer* primary and *integer* supplemental messages (*integer* total) after processing *integer* module(s)**  
**note** This message exists to provide some way of ensuring that an output message is always produced, even if there are no other messages. This is required for some windowing systems and can be useful to distinguish successful completion from premature termination. The message can also be useful for ensuring that all emitted messages have been accounted for. This message can be enabled with **+e900**.
- 901** **variable *symbol* of type *type* not initialized by definition**  
**note** The definition of the mentioned variable contained no initializer. While the use of an uninitialized variable is caught by warning 530, some style guidelines insist that variables should be initialized at definition. For example, see [11, Rule 19].  
**Supports CERT C EXP33-C - Do not read uninitialized memory**  
**Supports CWE-457 - Use of Uninitialized Variable**  
**Supports CWE-758 - Reliance on Undefined, Unspecified, or Implementation-Defined Behavior**  
**Supports CWE-908 - Use of Uninitialized Resource**
- 902** **non-static function *symbol* declared outside of a header**  
**note** A function declaration was found inside a source module and not in a header file. One common programming practice is to place all function declarations in headers.
- 904** **return statement before end of function *symbol***  
**note** A return statement was found before the end of a function definition. Many programming standards require that functions contain a single exit point located at the end of the function. This can enhance readability and may make subsequent modification less error prone.

Supports MISRA C 2012 Rule 15.5

Supports MISRA C++ Rule 6-6-5

Supports MISRA C 2004 Rule 14.7

### 905 non-literal format specifier used (with arguments)

**note** A `printf/scanf` style function received a non-literal format specifier but, unlike the case covered by Warning 592 the function also received additional arguments. E.g.

```
char *fmt;
int a, b;
...
printf( fmt, a, b );
```

Variable formats represent a very powerful feature of C/C++ but they need to be used judiciously. Unlike the case covered by Warning 592, this case cannot be easily rewritten with an explicit visible format. But this Elective Note can be used to examine code with non-literal formats to make sure that no errors are present and that the formats themselves are properly constructed and contain no user-provided data. See Warning 592.

Supports CWE-134 - *Use of Externally-Controlled Format String*

### 907 implicit conversion (*context*) from type *type* to type *type*

**note** A pointer whose type is not `void*` is being assigned to a variable (or passed to a parameter) whose type is `void*`. This is permitted in both C and C++. But the practice is potentially dangerous and this Elective Note allows one to see where this takes place. See also Note 908.

### 908 implicit conversion (*context*) from type *type* to type *type*

**note** A pointer whose type is `void*` is being assigned to a variable (or passed to a parameter) whose type is not `void*`. This conversion is not permitted in C++ (where Error message 64 is given in lieu of this message). But the conversion is permitted in C. Like the implicit conversion described by Message 907, the practice is potentially dangerous and this Elective Note allows one to see where this takes place.

Supports CERT C MEM02-C - *Immediately cast the result of a memory allocation function call into a pointer to the allocated type*

### 909 implicit boolean conversion from *type*

**note** A non-bool was tested as a Boolean. For example, in the following function:

```
int f(int n) {
    if (n) return n;
    else return 0;
}
```

the programmer tests 'n' directly rather than using an explicit Boolean expression such as 'n != 0'. Some shops prefer the explicit test.

### 910 implicit conversion of null pointer constant to pointer

**note** A pointer was assigned (or initialized) with a 0. Some programmers prefer other conventions such as `NULL` or `nil`. This message will help such programmers root out cavalier uses of 0. This is relatively easy in C since you can define `NULL` as follows:

```
#define NULL (void *)0
```

However, in C++, a `void*` cannot be assigned to other pointers without a cast. Instead, assuming that `NULL` is defined to be 0, use the option:

```
--emacro((910),NULL)
```

This will inhibit message 910 in expressions using NULL.

This method will also work in C. Both methods assume that you expressly turn on this message with a +e910 or equivalent.

**Supports AUTOSAR17 Rule A4-10-1**

**Supports AUTOSAR17 Rule M4-10-2**

**Supports AUTOSAR19 Rule A4-10-1**

**Supports AUTOSAR19 Rule M4-10-2**

**Supports MISRA C++ Rule 4-10-2**

### 911 implicit promotion from *type* to *type*

**note** Notes whenever a sub-integer expression such as a `char`, `short`, `enum`, or bit-field is promoted to `int` for the purpose of participating in some arithmetic operation or function call.

### 912 application of the usual arithmetic conversions converted a binary operand to a type other than that which would have been produced solely by application of integer promotion to that operand (text varies)

**note**

Notes whenever a binary operation (other than assignment) requires a type balancing. A smaller range type is promoted to a larger range type. For example: `3 + 5.5` will trigger such a message because `int` is converted to `double`.

### 915 implicit arithmetic conversion (*context*) from *type* to *type*

**note** Notes whenever an assignment, initialization or `return` implies an arithmetic conversion (*context* specifies which).

**Supports CERT C FLP34-C** - Ensure that floating-point conversions are within range of the new type

**Supports CERT C FLP36-C** - Preserve precision when converting integral values to floating-point type

**Supports CWE-197** - Numeric Truncation Error

**Supports CWE-681** - Incorrect Conversion between Numeric Types

### 916 implicit pointer assignment conversion (*context*) from *type* to *type*

**note** Notes whenever an assignment, initialization or `return` implies an implicit pointer conversion (*context* specifies which).

### 917 implicit conversion from *type* to *type* due to function prototype

**note** Notes whenever an implicit arithmetic conversion takes place as the result of a prototype. For example:

```
double sqrt(double);
... sqrt(3); ...
```

will elicit this message because 3 is quietly converted to `double`.

**Supports CERT C EXP47-C** - Do not call `va_arg` with an argument of the incorrect type

### 919 implicit conversion (*context*) from lower precision type *type* to higher precision type *type*

**note** A lower precision quantity was assigned to a higher precision variable as when an `int` is assigned to a `double`.

### 920 explicit cast from *type* to *type*

**note** A cast is being made to `void`.

**921 explicit cast from *type* to *type***

**note** A cast is being made from one integral type to another.

**922 explicit cast from *type* to *type***

**note** A cast is being made to or from one of the floating types (`float`, `double`, `long double`).

**Supports** CERT C FLP34-C - *Ensure that floating-point conversions are within range of the new type*

**Supports** CERT C FLP36-C - *Preserve precision when converting integral values to floating-point type*

**Supports** CWE-197 - *Numeric Truncation Error*

**Supports** CWE-681 - *Incorrect Conversion between Numeric Types*

**923 explicit cast from *type* to *type***

**note** A cast is being made from a pointer to a non-pointer or from a non-pointer to a pointer.

**Supports** MISRA C 2012 Rule 11.6

**Supports** MISRA C 2004 Rule 11.3

**925 explicit cast from *type* to *type***

**note** A cast is being made between pointers wherein the source or destination type is a pointer to `void`.

**926 explicit cast from *type* to *type***

**note** A cast is being made between pointers to (possibly `signed` or `unsigned`) `char`.

**927 explicit cast from *type* to *type***

**note** A cast is being made from pointer to (possibly `signed` or `unsigned`) `char` to a pointer to another type.

**928 explicit cast from *type* to *type***

**note** A cast is being made to pointer to (possibly `signed` or `unsigned`) `char` from a pointer to another type.

**929 explicit cast from *type* to *type***

**note** A cast is being made between pointers that does not fall under the purview of [920](#), [922](#), [923](#), [925](#), [927](#), [928](#).

**930 explicit cast from *type* to *type***

**note** A cast is being made to or from an enumeration type.

**931 both sides have side effects**

**note** Indicates when both sides of an expression have side-effects. An example is `n++ + f()`. This is normally benign. The really troublesome cases such as `n++ + n` are caught via Warning [564](#).

**Supports** CERT C EXP10-C - *Do not depend on the order of evaluation of subexpressions or the order in which side effects take place*

**Supports** MISRA C 2012 Rule 1.3

**Supports** MISRA C 2004 Rule 1.2

**935 data member *symbol* declared as type *type***

**note** This Note helps to locate non-portable data items within `struct`'s. If instead of containing `int`'s and

`unsigned int`'s, a `struct` were to contain `short`'s and `long`'s then the data would be more portable across machines and memory models. Note that bit fields and `union`'s do not get complaints.

**936 old-style function definition for function *symbol***

**note** An "old-style" function definition is one in which the types are not included between parentheses. Only names are provided between parentheses with the type information following the right parenthesis. This is the only style allowed by K&R.

**Supports** CERT C DCL07-C - *Include the appropriate type information in function declarators*

**Supports** MISRA C 2012 Rule 8.2

**937 old-style function declaration for function *symbol***

**note** An "old-style" function declaration is one without type information for its arguments.

**Supports** CERT C DCL20-C - *Explicitly specify void when a function accepts no arguments*

**Supports** MISRA C 2012 Rule 8.2

**Supports** MISRA C 2004 Rule 8.1

**Supports** MISRA C 2004 Rule 16.5

**940 omitted braces within initializer**

**note** An initializer for a subaggregate does not have braces. For example:

```
int a[2][2] = { 1, 2, 3, 4 };
```

This is legal C but may violate local programming standards. The worst violations are covered by Warning [651](#).

**Supports** AUTOSAR17 Rule M8-5-2

**Supports** AUTOSAR19 Rule M8-5-2

**Supports** MISRA C++ Rule 8-5-2

**Supports** MISRA C 2004 Rule 9.2

**941 result 0 due to operand(s) equaling 0 in operation '*operator*'**

**note** The result of a constant evaluation is 0 owing to one of the operands of a binary operation being 0. This is less severe than Info [778](#) wherein neither operand is 0. For example, expression `(2&1)` yields a [778](#) whereas expression `(2&0)` yields a [941](#).

**942 possibly truncated *string* promoted to *type***

**note** An integral expression (signed or unsigned) involving addition or subtraction is converted to a floating point number. If an overflow occurred, information would be lost. See also messages [647](#), [776](#) and [790](#).

**Supports** CERT C FLP06-C - *Convert integers to floating point for floating-point operations*

**943 too few initializers for aggregate *symbol* of type *type***

**note** The initializer `{0}` was used to initialize an aggregate of more than one item. Since this is a very common thing to do, it is given a separate message number, which is normally suppressed. See [785](#) for more flagrant abuses.

**944 left/right operand to '*operator*' always evaluates to '*true/false*'**

**note** The indicated operator (which is either `&&`, `||`, or `!`) has an argument that appears to always evaluate to either `true` or `false`. Information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message [506](#), which is based on testing constants or combinations of

constants.

Supports AUTOSAR17 Rule M0-1-1

Supports AUTOSAR17 Rule M0-1-2

Supports AUTOSAR17 Rule M0-1-9

Supports AUTOSAR19 Rule M0-1-1

Supports AUTOSAR19 Rule M0-1-2

Supports AUTOSAR19 Rule M0-1-9

Supports MISRA C++ Rule 0-1-1

Supports MISRA C++ Rule 0-1-2

Supports MISRA C++ Rule 0-1-9

Supports CWE-570 - *Expression is Always False*

Supports CWE-571 - *Expression is Always True*

#### 945 variable of undefined structure type declared to be extern

**note** Some compilers refuse to process declarations of the form:

```
extern struct X s;
```

where `struct X` is not yet defined. This note can alert a programmer porting to such platforms.

#### 946 relational operator *string* applied to pointers

**note** A relational operator (one of `>`, `>=`, `<`, `<=`) or the subtract operator has been applied to a pair of pointers. The reason this is of note is that when large model pointers are compared (in one of the four ways above) or subtracted, only the offset portion of the pointers is subject to the arithmetic. It is presumed that the segment portion is the same. If this presumption is not accurate then disaster looms. By enabling this message you can focus in on the potential trouble spots.

Supports AUTOSAR17 Rule M5-0-18

Supports AUTOSAR19 Rule M5-0-18

Supports MISRA C 2012 Rule 18.3

Supports MISRA C++ Rule 5-0-18

#### 947 pointer subtraction

**note** An expression of the form `p - q` was found where both `p` and `q` are pointers. This is of special importance in cases where the maximum pointer can overflow the type that holds pointer differences. For example, suppose that the maximum pointer is 3 Gigabytes -1, and that pointer differences are represented by a `long`, where the maximum `long` is 2 Gigabytes -1. Note that both of these quantities fit within a 32 bit word. Then subtracting a small pointer from a very large pointer will produce an apparent negative value in the `long` representing the pointer difference. Conversely, subtracting a very large pointer from a small pointer can produce a positive quantity.

The alert reader will note that a potential problem exists whenever the size of the type of a pointer difference equals the size of a pointer. But the problem doesn't usually manifest itself since the highest pointer values are usually less than what a pointer could theoretically hold. For this reason, the message cannot be given automatically based on scalar types and hence has been made an Elective Note.

Compare this Note with that of [946](#), which was designed for a slightly different pointer difference problem.

Supports AUTOSAR17 Rule M5-0-15

Supports AUTOSAR17 Rule M5-0-17

Supports AUTOSAR19 Rule M5-0-15

Supports AUTOSAR19 Rule M5-0-17

Supports MISRA C 2012 Rule 18.2

Supports MISRA C++ Rule 5-0-15

Supports MISRA C++ Rule 5-0-17

**948 operator *operator* always evaluates to *true/false***

**note** The operator named in the message is one of four relational operators or two equality operators in the list:

```
>   >=   <   <=
==  !=
```

The arguments are such that it appears that the operator always evaluates to either **true** or to **false** (as indicated in the message). This is similar to message 944. Indeed there is some overlap with that message. Message 944 is issued in the context where a Boolean is expected (such as the left hand side of a ? operator) but may not involve a relational operator. Message 948 is issued in the case of a relational (or equality) operator but not necessarily in a situation that requires a Boolean.

**Supports AUTOSAR17 Rule M0-1-9**

**Supports AUTOSAR19 Rule M0-1-9**

**Supports MISRA C++ Rule 0-1-9**

**Supports CWE-570 - Expression is Always False**

**Supports CWE-571 - Expression is Always True**

**952 parameter *symbol* of function *symbol* could be *const***

**note** A parameter is not modified by a function. For example:

```
int f( char *p, int n ) { return *p = n; }
```

can be redeclared as:

```
int f( char * const p, const int n ) { return *p = n; }
```

There are few advantages to declaring an unchanging parameter a **const**. It signals to the person reading the code that a parameter is unchanging, but, in the estimate of most, reduces legibility. For this reason the message has been given an Elective Note status.

However, there is a style of programming that encourages declaring parameters **const**. For the above example, this style would declare **f** as

```
int f( char *p, int n);
```

and would use the **const** qualifier only in the definition. Note that the two forms are compatible according to the standard. The declaration is considered the interface specification where **const** does not matter. The **const** does matter in the definition of the function, which is considered the implementation. Message 952 could be used to support this style.

Marking a parameter as **const** does not affect the type of argument that can be passed to the parameter. In particular, it does not mean that only **const** arguments may be passed. This is in contrast to declaring a parameter as pointer to **const** or reference to **const**. For these situations, Informational messages are issued (818 and 1764 respectively) and these do affect the kinds of arguments that may be passed. See also messages 953 and 954.

**Supports AUTOSAR17 Rule A7-1-1**

**Supports AUTOSAR19 Rule A7-1-1**

**Supports CERT C INT08-C - Verify that all integer values are in range**

**Supports MISRA C++ Rule 7-1-1**

**953 local variable *symbol* could be *const***

**note** A local auto variable was initialized and referenced but never modified. Such a variable could be declared **const**. One advantage in making such a declaration is that it can furnish a clue to the program reader that the variable is unchanging. Other situations in which a **const** can be added to a declaration are covered in messages 818, 843, 844, 952, 954 and 1764.

Supports AUTOSAR17 Rule A7-1-1

Supports AUTOSAR19 Rule A7-1-1

Supports CERT C DCL00-C - *Const-qualify immutable objects*

Supports MISRA C++ Rule 7-1-1

#### 954 local variable *symbol* could be pointer to `const`

**note** The data pointed to by a pointer is never changed (at least not through that pointer). It may therefore be better, or at least more descriptive, if the variable were typed pointer to `const`. For example:

```
{
    char *p = "abc";
    for( ; *p; p++ ) print(*p);
}
```

can be redeclared as:

```
{
    const char *p = "abc";
    for( ; *p; p++ ) print(*p);
}
```

It is interesting to contrast this situation with that of pointer parameters. The latter is given Informational status (818) because it has an effect of enhancing the set of pointers that can be passed into a function. Other situations in which a `const` can be added to a declaration are covered in messages 952, 953 and 1764.

Supports MISRA C 2012 Rule 8.13

#### 955 parameter *integer (type)* of forward declaration of *symbol* lacks a name

**note** In a function declaration a parameter name is missing. For example:

```
void f(int);
```

will raise this message. This is perfectly legal but misses an opportunity to instruct the user of a library routine on the nature of the parameter. For example:

```
void f(int count);
```

would presumably be more meaningful. [12, Rule 34]

This message is not given for function definitions, only function declarations.

Supports MISRA C 2012 Rule 8.2

Supports MISRA C 2004 Rule 16.3

#### 956 *string* variable *symbol* of type *type* is neither `const` nor `atomic`

**note** This check has been advocated by programmers whose applications are multi-threaded. Software that contains modifiable data of static duration is often non-reentrant. That is, two or more threads cannot run the code concurrently. By 'static duration' we mean variables declared static or variables declared external to any function. For example:

```
int count = 0;
void bump() { count++; }
int get_count() { return count; }
```

If the purpose is to obtain a count of all the `bump()`'s by a given thread then this program clearly will not do since the global variable `count` sums up the `bump()`'s from all the threads. Moreover, if the purpose of the code is to obtain a count of all `bump()`'s by all threads, it still may contain a subtle error (depending on the compiler and the machine). If it is possible to interrupt a thread between the access of `count` and the subsequent



store, then two threads that are `bump()`'ing at the same time, may register an increase in the `count` by just one.

Please note that not all code is intended to be re-entrant. In fact most programs are not designed that way and so this Elective Note need not be enabled for the majority of programs. If the program is intended to be re-entrant, all uses of non-const static variables should be examined carefully for non-reentrant properties.

#### 957 function *symbol* defined without a prototype in scope

**note** A function was defined without a prototype in scope. It is usually good practice to declare prototypes for all functions in header files and have those header files checked against the definitions of the function to assure that they match. This message will not be emitted for functions without external linkage.

The following function would have this message emitted, assuming there was no previous declaration of the function previously in the file, or in an included header file.

```
void fn(int x, int y) { } // note 957
```

If you are linting all the files of your project together such cross checking will be done in the natural course of things. For this reason this message has been given a relatively low urgency of Elective Note.

**Supports MISRA C 2012 Rule 8.4**

**Supports MISRA C 2004 Rule 8.1**

#### 958 padding of *integer* bytes needed to align *string* on a *integer* byte boundary

**note** This message is given whenever padding is necessary within a `struct` to achieve a required member alignment. *symbol* designates that which is being aligned. Consider:

```
struct A { char c; int n; };
```

Assuming that `int` must be aligned on a 4-byte boundary and assuming the size of a `char` to be 1, then this message will be issued indicating that there will be a padding of 3 bytes to align the number.

The alignment requirements vary with the compiler, the machine and, sometimes, compiler options. When separately compiled programs need to share data at the binary level it helps to remove any artificially created padding from any of the structures that may be shared.

**Supports CERT C EXP42-C - Do not compare padding data**

#### 959 nominal structure size of '*integer*' bytes is not a whole multiple of its alignment of '*integer*' bytes

**note** The alignment of a structure (or union) is equal to the maximum alignment of any of its members. When an array of structures is allocated, the compiler ensures that each structure is allocated at an address with the proper alignment. This will require padding if the size of the structure is not an even multiple of its maximum alignment. For example:

```
struct A { int n; char ch; } a[10];
```

Assuming the size and alignment of `int` is 4 then the size of each struct is 5 but its alignment is 4. As a result each struct in the array will be padded with 3 bytes.

Alignment can vary with the compiler and the machine. If binary data is to be shared by separately compiled modules, it is safer to make sure that all shared structures and unions are explicitly padded.

**Supports CERT C EXP42-C - Do not compare padding data**

#### 963 qualifier '*string*' follows a type; use `-fqb` to reverse the test

**note** The declarations in the following example are equivalent:

```
//lint +e963 report on qualifier-type inversion
extern const char *p;
extern char const *p; // Note 963
```

The qualifiers 'const' and 'volatile' may appear either before or after or even between other declaration specifiers. Many programmers prefer a consistent scheme such as always placing the qualifier before the type. If you enable 963 (using +e963) this is what you will get by default. The message will contain the word 'follows' rather than the word 'precedes'.

There is a diametrically opposite convention, viz. that of placing the qualifier after the type. As the message itself reminds the user, you will obtain the reverse test if you turn off the `fqb` (place qualifiers before types) flag. Thus

```
//lint -fqb turn off the Qualifiers Before types flag
//lint +e963 report on type-qualifier inversion
extern const char *p; // Note 963
extern char const *p;
```

Note that the use of this flag will cause 'follows' in the message to be replaced by 'precedes' and the alternative option mentioned within the 'use' clause is changed to its opposite orientation.

Dan Saks [2] and Vandevorde and Josuttis, [3], section 1.4, provide convincing evidence that this alternative convention is indeed the better one.

#### 967 header file '*string*' does not have a standard include guard

**note** The header file cited in the message does not have a *standard include guard* which is a common mechanism to protect against the repeated inclusion of headers. A *standard include guard* has one of the following forms:

```
#ifndef Name
#define Name
...
#endif
```

or

```
#if !defined(Name)
#define Name
...
#endif
```

with nothing but comments before and after this sequence and nothing but comments between the `#if/#ifndef` and the `#define`. The macro `Name` must not be defined when the header is included for the first time. It is standard practice in many organizations to always place include guards within every header.

If the header instead employs `#pragma once`, the text of the message will be appended with "('#pragma once' employed)", such instances can be suppressed by using `-estring(967,"#pragma once")`. Headers that use a modified version of the above header guard forms where `Name` is defined with a value, e.g.:

```
#ifndef Name
#define Name 1
...
#endif
```

are not considered to be *standard include guard* but are recognized as *alternate include guards*. In such cases, the text of this message is appended with "('alternate include guard' employed)" and may be suppressed using `-estring(967,"alternate include guard")`. See also message 451.

Supports AUTOSAR17 Rule M16-2-3

Supports AUTOSAR19 Rule M16-2-3

**Supports CERT C PRE06-C** - *Enclose header files in an include guard*

**Supports MISRA C++ Rule 16-2-3**

### 970 use of modifier or type '*name*' outside of a typedef

**note** Some standards require the use of type names (defined in `typedef`'s) in preference to raw names used within the text of the program. For example they may want you to use `INT32` rather than `int` where `INT32` is defined as:

```
typedef int INT32;
```

This message is normally issued for the standard intrinsic types: `bool`, `char`, `wchar_t`, `int`, `float`, `double`, and for modifiers `unsigned`, `signed`, `short` and `long`. You may enable this message and then suppress the message for individual types to obtain special effects. For example, the following will enable the message for all but `bool`.

```
+e970 -estring(970,bool)
```

**Supports MISRA C 2012 Dir 4.6**

**Supports MISRA C++ Rule 3-9-2**

**Supports MISRA C 2004 Rule 6.3**

### 971 use of plain char

**note** The '`char`' type was specified without an explicit modifier to indicate whether the `char` was `signed` or `unsigned`. The plain `char` type can be regarded by the compiler as identifying a `signed` or an `unsigned` quantity, whichever is more efficient to implement. Because of this ambiguity, some standards do not like the use of `char` without an explicit modifier to indicate its signedness.

### 972 unusual character '*string*' in '*kind*' literal

**note** An unusual character was found in a character or string literal. It is identified in the message by its hexadecimal encoding. Characters are considered to be unusual if they are outside the standard C and C++ source character set. For example:

```
char ch = '\'; // Unusual character '\x60'
```

The backtick character being assigned above is considered unusual. To suppress this message for this character use the option:

```
-estring( 972, "\x60" )
```

### 973 unary operator in macro '*string*' not parenthesized

**note** A unary operator appearing in an expression-like macro was found to be not parenthesized. For example:

```
#define N -1
```

The user may prefer to parenthesize such things as:

```
#define N (-1)
```

This has been placed in the elective note category because we cannot find an instance when this really produces a problem. The important case of unparenthesized binary operators is covered with message [773](#).

**Supports CERT C PRE02-C** - *Macro replacement lists should be parenthesized*

**974 worst case stack usage: *detail***

**note** This message, issued at global wrap-up, will report on the function that requires the most stack. The stack required consists of the amount of auto storage the function requires plus the amounts required in any chain of functions called. The worst case chain is always reported.

To obtain a report of all the functions, use the `+stack` option.

Reasonable allowances are made for function call overhead and the stack requirements of external functions. These assumptions can be controlled via the `+stack` option.

If recursion is detected it will be reported here, as this is considered worse than any finite case. The next worst case is that the stack can't be determined because a function makes a call through a function pointer. The function is said to be non-deterministic. If neither of these conditions prevail, the function that heads the worst case chain of calls will be reported upon.

The message will normally provide you with the name of a called function. If the function is recursive this will provide you with the first call of a recursive loop. To determine the full loop, you will need a full stack report as obtained with the `+stack` option. You need a suboption of the form `&file=file` to specify a file that will contain a record for each function for which a definition was found. You will be able to follow the chain of calls to determine the recursive path.

If you can assure yourself through code analysis that there is an upper bound to the amount of stack utilized by some recursive function, then you can employ the `+stack` option to specify the bound for this function. The function will no longer be considered recursive but rather finite. In this way, possibly through a sequence of options, you can progressively eliminate apparent recursion and in that way arrive at a safe upper bound for stack usage. Similar considerations apply for non-deterministic functions.

**975 unknown pragma '*string*' will be ignored**

**note** The first identifier after `#pragma` is considered the name of the pragma. If the name is unrecognized then the remainder of the line is ignored. Since the purpose of `#pragma` is to allow for compiler- dependent communication, it is not really expected that all pragmas will be understood by all third-party processors of the code. Thus, this message does not necessarily indicate that there is anything wrong and could easily be suppressed entirely.

Moreover, if the pragma occurs in a library header, this message would not normally be issued because the option `-wlib(1)` would be in effect (this option is present in all of our compiler options files).

But, if the pragma occurs in user code, then it should be examined to see if there is something there that might interest a lint processor. There are a variety of facilities to deal with pragmas; in particular, they can be mapped into linguistic constructs or lint options or both. See Section [4.12.5 Pragmas](#).

**Supports MISRA C 2004 Rule 3.4**

**977 non-literal non-Boolean of type *type* used in a Boolean *string***

**note** This message is issued when a non-literal expression of non-boolean type is assigned to a boolean. This can occur through direct assignment, initialization, as an argument in a function call, or a return expression. For example, the function below returns true if there is a remainder when x is divided by y but the type of the value before conversion is `int`:

```
_Bool foo(int y, int z) {
    return y % z;    // Note 977
}
```

The message won't be issued for:

```
_Bool foo(int y, int z) {
    return y % z != 0;  // Okay, != implies boolean context
}
```

A cast can also be used to suppress this message.

### 978 the name '*name*' matches a pattern reserved to the compiler *string*

**note** The C Standard specifies a variety of naming patterns reserved for future use. For example, names starting with *is*, *to*, or *str* followed by a lowercase letter are reserved to the implementation. This message reports on symbols declared with names that match one of these patterns. The name of the offending symbol is provided along with the reserved pattern that the name matches. For example:

```
int strmin;
```

will elicit:

```
note 978: the name 'strmin' matches a pattern reserved to the compiler because
         it begins with 'str' and a following lowercase letter
int strmin;
^
```

**Supports CERT C DCL37-C** - Do not declare or define a reserved identifier

### 979 function *symbol* could be marked with a 'pure' semantic

**note** The specified function was analyzed and determined to be eligible for the `pure` semantic but no `-sem` option was used to specify that this function was `pure`. Since functions are considered to be impure by default when the definition is not visible from the current module, specifying this function as `pure` could improve analysis related to side-effects and purity.

### 980 macro name '*name*' matches a pattern reserved to the compiler *string*

**note** The C Standard specifies a variety of macro naming patterns reserved by the implementation. These patterns include a name starting with 'E' followed by a digit or upper case letter, names starting with 'SIG' or 'SIG\_' followed by an uppercase letter, and macros that begin with 'PRI' or 'SCN' followed by either 'X' or a lowercase letter. The message includes both the name of the offending macro and the reserved pattern that it matches. For example:

```
#define LC_END
```

will be greeted with:

```
note 980: macro name 'LC_END' matches a pattern reserved to the compiler
         because it begins with 'LC_' and a following uppercase letter
#define LC_END
^
```

Note that some patterns are reserved only in certain versions of C and will be diagnosed only when the language mode specified corresponds to the version in which the pattern is applicable. For example, names starting with `INT` and ending in `_C` are diagnosed only in C99 and later.

**Supports MISRA C 2004 Rule 20.1**

### 981 cast of expression of type *type* to same type is redundant

**note** A cast is being performed on an expression that is already of the type being cast to making the cast redundant. This message is not issued for casting enumerations to their underlying type or within template instantiations.

**983 behavior of dash in scan list is implementation defined**

**note** A dash (-) was encountered within the scan list in a %[ conversion specifier in the call to a `scanf`-like function. Furthermore the dash was not the first character (or the second character where the first character is ^) or the last character, e.g. %[A-Z]. The behavior of a dash in this position is implementation defined, some implementations interpret this as a range of characters to include in the scan set (e.g. all characters from A to Z) while others treat it literally.

**986 target type *type* of type alias *symbol* is deprecated**

**note** This message is issued when a type that has been deprecated using the `-deprecate` option with a category of *type* is used as a target in a `typedef` definition. This is to provide notification that the underlying deprecated type may be used through a `typedef` later, which will not be diagnosed. To diagnose uses of a type through a `typedef`, the `basetype` deprecation category can be used. See the `-deprecate` option for more information.

**987 constructor parameter *symbol* shadows the field *symbol* of *symbol***

**note** This message is a weaker form of message 578 for cases where a field is shadowed by a constructor parameter.

**999 defaulting to *string* concurrent threads**

**note** The `-max_threads=n` option can be used to specify the number of concurrent linting threads for parallel analysis. If *n* is specified as 0, PC-lint Plus attempts to query the hardware to determine the optimal value for *n*. This message serves to inform the programmer of the value that was selected.

## 22.3 Messages 1000-1999

**1001 scope *string* must be a class*string***

**error** In an expression of the form `X::Y`, *X* must be a class name. [13, Section 10.4]

**1002 invalid use of 'this' *context***

**error** The keyword `this` refers to the class being passed implicitly to a member function. It is invalid outside a class member function. [13, Section 5.1]

**1003 'this' cannot be used in a static member function declaration**

**error** A static member function receives no `this` pointer. [13, Section 9.5]

**1004 right hand operand to '*string*' has non-pointer-to-member type *type***

**error** The `.*` and `->*` operators require pointer to members on the right hand side. [13, Section 5.5]

**1008 initializer on function does not look like a pure-specifier**

**error** Some nonstandard extensions to C++ allow integers to follow '=' for declarations of member functions. If you are using such extensions, simply suppress this message. If only library headers are using this extension, use `-elib(1008)`. [13, Section 10.3]

**1012 return type not allowed for conversion function**

**error** The return type of a function introduced with '`operator type`' is *type* and may not be preceded with the same or any other type. [13, Section 12.3.2]

**1013 symbol *symbol* is not a member of *string***

**error** The second operand of a scope operator or a `'.'` or `'->'` operator is not a member of the `class` (`struct` or `union`) expressed or implied by the left hand operand. [13, Section 3.2]

**1020 template specialization declared without a 'template<>' prefix**

**error** A class template specialization is generally preceded by a `'template'` clause as in:

```
template< class T > class A { };    // a template
template<> class A<int> { };       // a specialization
```

If the `'template<>'` is omitted you will get this message but it will still be interpreted as a specialization. Before the standardization of template syntax was completed, a template specialization did not require this clause and its absence is still permitted by some compilers.

**1022 function *symbol* must be a non-static member function**

**error** There are four operators not to be defined except as class members. These are:

```
= () [] ->
```

The parameter *symbol* indicates which it is. [13, Section 13.4.3 and 13.4.6]

**1023 call to *symbol* is ambiguous**

**error** A call to an overloaded function or operator is ambiguous. The candidates of choice are provided in the message. [13, Section 13.2]

**1024 no matching function for call to *symbol***

**error** A call to an overloaded function could not be resolved successfully because no function is declared with the same number of arguments as in the call. [13, Section 13.2]

**1027 missing default argument for parameter *symbol* of function *symbol***

**error** Default arguments need to be consecutive. For example

```
void f(int i=0, int j, int k=0);
```

is illegal. [13, Section 8.2.6]

**1029 default argument repeated for parameter *symbol* in function *symbol***

**error** A default value for a given argument for a given function should be given only once. [13, Section 8.2.6]

**1031 local variable *symbol* used in default argument expression**

**error** Default values for arguments may not use local variables. [13, Section 8.2.6]

**1032 *non-static/explicit* function *symbol* cannot be called without object**

**error** There was an attempt to call a non-static member function without specifying or implying an object that could serve as the basis for the `this` pointer. If the member name is known at compile time it will be printed with the message. [13, Section 5.24]

- 1033 static member functions cannot be virtual**  
**error** You may not declare a static member function `virtual`. [13, Section 10.2]
- 1034 'static' can only be specified inside the class definition**  
**error** This can come as a surprise to the novice C++ programmer. The word '`static`' within a class definition is used to describe a member that is alone and apart from any one object of a class. But such a member has program scope not file scope. The word '`static`' outside a class definition implies file scope not program scope. [13, Section 9.5]
- 1036 call to constructor of *symbol* is ambiguous**  
**error** There is more than one constructor that can be used to make a desired conversion. [13, Section 12.3.2]
- 1037 conversion between *types* is ambiguous**  
**error** There is more than one conversion function (of the form `operator type ()`) that will perform a desired conversion. [13, Section 12.3.2]
- 1038 type *type* not found, *type* assumed**  
**error** We have found what appears to be a reference to a type but no such type is in scope. We have, however, been able to locate a type buried within another class. Is this what the user intended? If this is what is intended, use full scoping. If your compiler doesn't support the scoping, suppress with `-esym`. [13, Section 3.2]
- 1040 non-friend class member *symbol* cannot have a qualified name**  
**error** A declaration of the symbol `X::Y` appears within a class definition (other than for class `X`). It is not a `friend` declaration. Therefore it is in error.
- 1042 at least one class-like parameter is required for overloaded *string***  
**error** In defining (or declaring) an operator you must have at least one class as an operand. [13, Section 13.4]
- 1043 cannot delete expression of type *type***  
**error** An expression being `delete`d is a non-pointer, non-array. You may only `delete` what was created with an invocation of `new`. [13, Section 5.3.4]
- 1046 invalid use of member *symbol* in *static/explicit object* member function**  
**error** The *symbol* is a non-static member of a class and hence requires a class instantiation. None is in sight. [14, Section `class.static`]
- 1051 redefinition of *symbol* as different kind of symbol**  
**error** Whereas it is possible to overload a function name by giving it two different parameter lists, it is not possible to overload a name in any other way. In particular, a function name may not also be used as a variable name. [13, Section 9.2]  
**Supports CERT C DCL36-C - Do not declare an identifier with conflicting linkage classifications**
- 1055 use of undeclared identifier *string*; did you mean *symbol*?**  
**error** Whereas in C you may call a function without a prior declaration, in C++ you must supply such a declaration.



For C programs you would have received an Informational message (718) in this event. [13, Section 5.2.2]

**1057 member *symbol* cannot be used without an object**

**error** The indicated member referenced via scope operator cannot be used in the absence of a **this** pointer. [13, Section 5.2.4]

**1063 copy constructor for class *symbol* must pass its first argument by reference**

**error** A constructor for a class closely resembles a copy constructor. A copy constructor for class **X** is typically declared as:

```
X(const X &)
```

If you leave off the '&' then a copy constructor would be needed just to copy the argument into the copy constructor. This is a runaway recursion. [13, Section 12.1]

**1069 member initializer *string* does not name a non-static data member or base class of class *symbol***

**error** Within a constructor initialization list, a name was found that did not correspond to either a direct base class of the class being defined or a member of the class.

**1071 constructor cannot have a return type**

**error** Constructors and destructors may not be declared with a return type, not even **void**. See ARM [13, Section 12.1 and 12.4]

**1072 reference variable *string* must be initialized**

**error** A reference variable must have an initializer at the point of declaration.

**1074 expected a namespace identifier**

**error** In a declaration of the form:

```
namespace name = scoped-identifier
```

the *scoped-identifier* must identify a namespace.

**1075 ambiguous reference to symbol *symbol***

**error** Two namespaces contain the same name. A reference to such a name could not be disambiguated. You must fully qualify this name in order to indicate the name intended.

**1076 anonymous union assumed to be 'static'**

**error** Anonymous unions need to be declared **static**. This is because the names contained within are considered local to the module in which they are declared.

**1079 could not find '>' or ',' to terminate template parameter**

**error** The default value for a template parameter appears to be malformed. For example, suppose the user mistakenly substituted a ']' for a '>' producing the following:

```
template <class T = A< int ] >
    class X
```

```
{
};
```

This will cause PC-lint Plus to process to the end of the file looking (in vain) for the terminating pointy bracket. Not finding it will cause this message to be printed. Fortunately, the message will bear the *location* of the malformed template.

**1083 ambiguous conversion between 2nd and 3rd arguments of conditional operator; *reason***

**error** If the 2nd operand can be converted to match the type of the 3rd, and the 3rd operand can be converted to match the type of the 2nd, then the conditional expression is considered ill-formed.

**1088 a using declaration requires a qualified-id**

**error** This error is issued when a using-declaration references a name without the `::` scope resolution operator; e.g.:

```
class A {
protected:
    int n;
};

class B : public A {
public:
    using n; // Error 1088: should be 'using A::n;'
};
```

See [4, Section 7.3.3 namespace.udecl].

**1089 a using declaration must not name a namespace**

**error** This error is issued when the rightmost part of the qualified-id in a using-declaration is the name of a namespace. E.g.:

```
namespace N {
    namespace Q {
        void g();
    }
}

void f() {
    using ::N::Q; // Error 1089
    Q::g();
}
```

Instead, use a namespace-alias-definition:

```
namespace N {
    namespace Q {
        void g();
    }
}

void f() {
    namespace Q = ::N::Q; // OK
    Q::g();               // OK, calls ::N::Q::g().
}
```

See [10], Issue 460.

**1090 a using declaration must not name a template-id**

**error** This error is issued when the rightmost part of the qualified-id in a using-declaration is a template-id. E.g.:

```
template <class T> class A {
protected:
    template <class U> class B {};
};

struct D : public A<int> {
public:
    using A<int>::B<char *>; // Error 1090
};

D::B<char *> bc;
```

Instead, refer to the template name without template arguments:

```
template <class T> class A {
protected:
    template <class U> class B {};
};

struct D : public A<int> {
public:
    using A<int>::B; // OK
};

D::B<char *> bc;    // OK
```

See [4], 7.3.3 namespace.udecl.

**1091 using declaration refers into *symbol*, which is not a base class of *symbol***

**error** This error is issued when the nested-name-specifier of the qualified-id in a using-declaration does not name a base class of the class containing the using-declaration; e.g.:

```
struct N {
    void f();
};

class A {
protected:
    void f();
};

class B : A {
public:
    using N::f;    // Error 1091
};
```

See [10], Issue 400.

**1092 a using declaration that names a class member must be a member declaration**

**error** This error is issued when the nested-name-specifier of the qualified-id in a using-declaration names a class but the using-declaration does not appear where class members are declared. E.g.:

```

    struct A {
        void f();
    };

    struct B : A {
        void g() { using A::f; }    // Error 1092
    };

```

See [4], 7.3.3 namespace.udecl.

**1093 *symbol* is not virtual and cannot be declared pure**

**error** A pure specifier ("= 0") should not be placed on a function unless the function had been declared "virtual".

**1096 an initializer for a delegating constructor must appear alone**

**error** C++11 requires that if a constructor delegates to another constructor, then the *mem-initializer* (the region between the colon and the function body) must contain only one item, and that item must be a call to another constructor (which is called the "target constructor"). Example

```

    struct A {
        int n;
        A(int);
        A(const A &p) : A(p.n) { }    // OK
        A() : n(42), A(32) { }        // Error 1096
    };

```

**1097 constructor *symbol* creates a delegation cycle**

**error** The specified constructor is a delegating constructor that contains a delegation cycle, either directly by delegating to itself or indirectly by calling another delegating constructor that eventually delegates back to the original constructor. If multiple constructors are in the cycle, the other constructors are provided via subsequent [891](#) supplemental messages. For example:

```

    struct A {
        int n;
        A(int x) : A(x, 0) { }        // Error 1097
        A(int x, int y) : A(x, y, 0) { }
        A(int x, int y, int z) : A(x) { }
    };

```

**1098 function template specialization *symbol* does not match any function template**

**error** This message is issued for a declaration where the user apparently intended to name a specialization of a function template (e.g., in an explicit specialization, an explicit instantiation or a friend declaration of specialization), but no previously-declared function template is matched. Example:

```

    template<class T> void f( const T& ); // #1

    struct A{};
    template<> void f( const A& );        // Ok
    // (A is the deduced argument to T.)

    struct B{};
    template<> void f( const B );        // Error 1098.
    // (A template argument cannot be deduced for T.)

```

**1099** function template specialization *symbol* ambiguously refers to more than one function template; explicitly specify template arguments to identify a particular function template

error

This message is issued for a declaration where the user apparently intended to name a specialization of a function template (e.g., in an explicit specialization, an explicit instantiation or a friend declaration of specialization), but the specialization matches multiple function templates, and none of the matched templates is more specialized than all of the other matching templates. The candidates (i.e., the matching templates) are provided in the message. Example:

```
template<class T> struct A {};

template<class T, class U> void f( T*, U ); // #1
template<class T, class U> void f( T, A<U> ); // #2

struct B{};
template<> void f( B, A<B> ); // Ok
// #1 does not match but #2 does.

template<> void f( char*, A<int> ); // Error 1099
// Both #1 and #2 match and neither is more specialized than the other.
```

This situation can be avoided in at least a couple of ways. One way is to explicitly specify one or more template arguments. Example

```
// continuing from above...
template<> void f<char*>( char*, A<int> ); // Ok
// #1 does not match but #2 does.
```

Another way is to use SFINAE (Substitution Failure Is Not An Error) tactics in the declaration of one or more function templates, e.g. with `boost::enable_if`.

**1101** type of variable *symbol* cannot be deduced from its initializer

error Example:

```
int f(void);
int f(char*);
auto n = f; // Error
```

In terms of deduction, this is equivalent to:

```
int f(void);
int f(char *);
template <class T> void g(const T &);

void h(void) {
    g(f); // Error
}
```

Here, 'f' refers to multiple overloaded functions, so it is an ambiguous reference and T cannot be deduced. (Code like this could still be well-formed however, e.g. if g is overloaded with a non-template function whose parameter type is 'ptr-to-function returning int taking char\*').

**1102** *string*' type deduced inconsistently: *type* for *symbol* but *type* for *symbol*

error

When multiple variables are defined in the same declaration, and when that declaration uses the keyword `auto` as the *type-specifier*, the type for which `auto` is a placeholder must be the same for each variable. Example:

```

float g(void);
char* s();
auto a = 42;           // Ok, auto is 'int'
auto b = g();          // Ok, auto is 'float'
auto c = 'q',
    *d = s();          // Ok, auto is 'char' (for both c and d)
auto x = 42, y = g();  // Error 1102 here

```

**1103 non-integral/bit-precise type type is not a valid enum-base**

**error** When an enumeration type is declared with an explicit underlying type, that type must be integral. It must not be a bit-precise integral type. Example:

```

enum A : bool;          // ok
enum B : short;         // ok
enum C : unsigned long long; // ok
enum D : float;         // error 1103
enum E : _BitInt(4);    // error 1103

```

**1105 cannot overload a member function *with a certain ref-qualifier* with a member function *with different ref-qualifiers***

**error** If an explicit ref qualifier ('&' or '&&') of a nonstatic member function is employed, an explicit ref qualifier needs to be used with every member of the overload set. Thus:

```

class A {
    void f(int)&;
    void f(int);
    void f(double);
    void g(int);
    void g(double);
};

```

**1107 invalid concatenation of wide string literals of different kinds**

**error** Two string literals that different types are being concatenated. Examples:

```

char *s = u"abc" U"def";
char *q = L"ghi" u"jkl";

```

This message is issued for mixing strings of `char16_t`, `char32_t`, and/or `wchar_t` (as shown). Literal string concatenation of any of these with an ordinary character literal is permitted and will receive Informational [707](#).

Supports AUTOSAR17 Rule A2-14-2

Supports AUTOSAR19 Rule A2-13-2

**1108 call to deleted function *string***

**error** This message is issued when an attempt is made to call a deleted function. For example:

```

void f( int ) = delete;
void f( double );
void g( double d, int n ) {
    f( d ); // Ok
    f( n ); // Error
}

```

**1110 circular pointer delegation detected: explicit application of *type::operator->* causes infinite applications of the same operator**

error

When an overloaded `operator->` is used as in

```
a->b
```

it is effectively expanded to:

```
a.operator->()->b
```

And this expansion repeats until an `operator->` is found that does not yield a class type. But in the process of evaluating this expansion, it might be found that one of the operators returns a class type for which an overloaded `operator->` was already expanded; in that case, Error 1110 is triggered. Example:

```
struct B;
struct A { struct B& operator->(); };
struct B { struct A& operator->(); };
int f( A & p ) { p->g(); }           // Error
```

**1112 function with trailing return type must specify return type 'auto', not *string***

error

or example, if you want to declare that `f` returns a pointer-to-int, you must write:

```
auto f() -> int *;
```

... and not:

```
auto *f() -> int;
```

This also applies to a `type-id` (e.g., in a cast to a pointer-to-function, or as an argument to a template type-parameter).

**1116 virtual function *symbol* overrides function marked with '*string*'**

error

A derived class attempted to override a virtual function that is marked with the `final` virt-specifier in a base class.

**1117 non-virtual function *symbol* marked with '*string*'**

error

A virt-specifier (`final` or `override`) was supplied to a non-virtual function.

**1118 virtual function already marked '*string*'**

error

A virt-specifier (`final` or `override`) was encountered multiple times for the specified virtual function.

**1119 virtual function *symbol* marked 'override' does not override any member functions**

error

A virtual function was marked with the `override` keyword but does not override a base class function.

**1120 incomplete type *type* is not a valid range expression**

error

An incomplete type was used as a range expression in a range-based `for` statement. A range expression must be a complete type.

**1123 attempt to derive from class *type* marked as '*final/sealed*'**

error

A class that was marked with the `final` class-virt-specifier was used as a base class in a class declaration.

**1124 digit separator not allowed: before/after digit sequence**

**error** A digit separator character was encountered within a numeric literal at a point where digit separators are not allowed. Digit separators are only allowed between digits of a numeric literal and cannot be adjacent to each other.

**1125 a type cannot be defined in a friend declaration**

**error** The definition of a type appeared in a friend declaration, this is not legal.

Example:

```
class A {
    friend struct B;    // ok
    friend struct C {}; // error
};
```

**1127 catch handler after catch(...)**

**error** A `catch` handler appeared following a `catch(...)` in the same `try-catch` statement, which invokes undefined behavior.

Supports AUTOSAR17 Rule M15-3-7

Supports AUTOSAR19 Rule M15-3-7

Supports MISRA C++ Rule 15-3-7

**1301 integer sequences must have non-negative sequence length**

**error** An attempt was made to instantiate the built-in template `__make_integer_seq` with a negative length.

**1302 integer sequences must have integral element type**

**error** An attempt was made to instantiate the built-in template `__make_integer_seq` with a non-integral type. As the name implies, `__make_integer_seq` generates a sequence of *integers*.

**1401 non-static data member *symbol* not initialized by constructor *symbol***

**warning** The indicated non-static data member was not initialized by the specified constructor. Specifically, this means that the member does not have an in-class initializer and was neither directly initialized or assigned a value in the constructor nor did the constructor call any function that appeared to initialize this member.

Supports AUTOSAR17 Rule A12-1-1

Supports AUTOSAR19 Rule A12-1-1

Supports CWE-456 - *Missing Initialization of a Variable*

**1404 deleting an object of type *type* before that type is defined**

**warning** The following situation was detected:

```
class X; ... X *p; ... delete p;
```

That is, a placeholder declaration for a class is given and an object of that type is deleted before any definition is seen. This may or may not be followed by the actual class definition:

```
class X { ... };
```

A `delete` before the class is defined as dangerous because, among other things, any `operator delete` that may be defined within the class could be ignored.

Supports AUTOSAR19 Rule A5-3-3



- 1405 header <typeinfo> must be included before 'typeid' is used**  
**warning** According to Section 5.2.8 (para 6) of the C++ standard [14], "If the header <typeinfo> (18.5.1) is not included prior to a use of `typeid`, the program is ill-formed." A `typeid` was found in the program but the required include was not.
- 1407 incrementing expression of type bool**  
**warning** An increment operator was applied to an object of `bool` type; such use has been deprecated since C++98. The same effect can be obtained by assigning the value `true` to the object. Note the decrementing an object of `bool` type has never been allowed in Standard C++ and will instead be greeted with an error.  
**Supports AUTOSAR17 Rule A1-1-1**  
**Supports AUTOSAR19 Rule A1-1-1**
- 1411 member *symbol* with different signature hides virtual member *symbol***  
**warning** A member function has the same name as a virtual member of a derived class but it has a different signature (different parameter list). This is legal but suspicious because it looks as though the function would override the virtual function but doesn't. You should either adjust the parameters of the member so that the signatures conform or choose a different name. See also message [1511](#).
- 1413 function *symbol* is returning a temporary via a reference**  
**warning** It appears that a function (identified as *symbol* in the message) declared to return a reference is returning a temporary. The C++ standard (Section 12.2), in addressing the issue of binding temporary values to references, says "A temporary bound to the returned value in a function return statement ... persists until the function exits". Thus the information being returned is not guaranteed to last longer than the function being called.
- It would probably be better to return by value rather than reference. Alternatively, you may return a static variable by reference. This will have validity at least until the next call upon the same function.
- 1414 assigning address of local variable *symbol* to member of 'this' object**  
**warning** The address of an auto variable was taken and assigned to a `this` member in a member function. For example:
- ```

struct A {
    char *x;
    void f() {
        char y[10];
        x = y;          // warning 1414
    }
};

```
- Here, the address of `y` is being passed to member `x` but this is dangerous (if not ridiculous) since when the function returns the storage allocated for `y` is deallocated and the pointer could very easily harm something.
- 1415 pointer to non-POD *type* passed to function *symbol* (*context*)**  
**warning** A non-POD class is one that goes beyond containing just Plain Old Data (POD). In particular, it may have private or protected data or it may have constructors or a destructor or a copy assignment. All of these things disqualify it from being a POD. A POD is fully defined in the C++ standard (Clause 9).

Some functions (such as `memcpy`, `memcmp`, `memmove`, etc.) are expected to be given only pointers to POD objects. The reason is that only POD objects have the property that they can be copied to an array of bytes and back again with a guarantee that they will retain their original value. (See Section 3.9 of the C++

standard [4]). See also Semantic `pod(i)`.

**Supports AUTOSAR19 Rule A12-0-2**

**Supports CWE-687 - Function Call With Incorrectly Specified Argument Value**

**1416 reference *symbol* is not yet bound to a value when used here**

**warning** This message is usually issued when a reference to a member of a class is used to initialize a reference to another member of the same class before the first member was initialized. For example:

```
class C {
    int &n, &m;
    C(int &k) : n(m), m(k) { /* ... */ }
};
```

Here `m` is initialized properly to be identical to `k`. However, the initialization of `n`, taking place, as it does, before `m` is so initialized, is erroneous. It is undefined what location `n` will reference.

**Supports CWE-457 - Use of Uninitialized Variable**

**1417 *string* must explicitly initialize the *reference/const* member *symbol***

**warning** This message is issued when a reference data member of a class does not appear in a mem-initializer. For example, the following code will result in a Warning 1417 for symbol `m` since a mem-initializer is the only way that `m` can be reference initialized.

```
class C {
    int &n, &m;
    C(int &k) : n(k) { /* ... */ }
};
```

**1419 throwing the NULL macro will invoke an implementation-defined handler; NULL may be defined as either an integer literal equal to zero or the keyword nullptr**

**warning** The macro `NULL` was passed to a `throw` expression. Since C++11, the `NULL` macro can be defined as expanding to either an integer literal with a zero value or `nullptr`, the choice of which is implementation defined. The handler that catches the exception may depend on how the `NULL` macro is defined. Prior to C++11, `NULL` could only be defined as an integer type and will not be caught by an exception handler expecting a pointer type, which may not be obvious.

**Supports AUTOSAR17 Rule M15-1-2**

**Supports AUTOSAR19 Rule M15-1-2**

**Supports MISRA C++ Rule 15-1-2**

**1420 'mutable' applied to a reference type is non-standard**

**warning** C++ expressly forbids the use of the `mutable` keyword on a reference type. Despite this, at least one vendor's compiler not only supports this use but relies on the ability to do so in their own library headers. If your compiler supports this use, you can suppress this message.

**1421 template parameter illegally redefines default argument**

**warning** C++ explicitly forbids redefining the default argument of a template parameter. If your compiler allows this, you can suppress this message.

**1422 default constructor *symbol* defaulted outside of class**

**warning** This message is issued when the default constructor for a class is declared within its class definition and then

defaulted outside of the class definition. C++ performs initialization of classes with user supplied default constructors (declared but not defaulted in the class definition) differently than classes without a user provided default constructor.

**1423** **reinterpret\_cast from *type* to *type* has undefined behavior**  
**warning** The C++ Standard specifies the conversions that can be performed using `reinterpret_cast`. A conversion was attempted using `reinterpret_cast` that was not included in this list which will result in undefined behavior.

**1501** **data member *symbol* has zero size**  
**warning** A data member had zero size. It could be an array of zero length or a class with no data members. Check your code to make sure this is not an error. Some libraries employ clever templating, which will elicit this message. In such a case it is necessary for you to inhibit the message outright (using `-e1501`) or through a judicious use of `-esym(1501,...)`.

```
struct A {
    int b[0];    // warning 1501
};
```

**1502** **defined object *symbol* of type *symbol* has no non-static data members**  
**warning** A variable is being instantiated that belongs to a class that contains no data members (either directly or indirectly through inheritance). Note that this message can be suppressed using `-esym` with either the object name (given in the first *symbol* parameter) or the class name (given in the second *symbol* parameter). [13, Section 9]

**1504** **anonymous structure declaration without the '+fas' option**  
**warning** An untagged struct declaration appeared within a `struct/union` and has no declarator. It is not treated like an anonymous union. Was this intended?

**1505** **base specifier with no access specifier is implicitly *public/private***  
**warning** A base class specifier provides no access specifier (`public`, `private` or `protected`). An explicit access specifier is always recommended since the default behavior is often not what is expected. For example:

```
class A : B { int a; };
```

would make B a private base class by default.

```
class A : private B { int a; };
```

is preferred if that's what you want. [13, Section 11.1]

**1506** **call to virtual function *symbol* within a *string***  
**warning** A call to a virtual function was found in a constructor or a destructor of a class; such a call will not consider overrides from derived classes (as they have not been constructed yet, or have already been destroyed). This message will not be issued in any of the following cases:

- The call was qualified explicitly using the scope operator, inhibiting the virtual call mechanism.
- The virtual function was declared with the final specifier.
- The class of the constructor or destructor was declared with the final specifier.

[15, Section 9]

**Supports AUTOSAR17 Rule M12-1-1**

**Supports AUTOSAR19 Rule M12-1-1**

**Supports MISRA C++ Rule 12-1-1**

- 1507** **use of 'delete' on pointer-to-array type *type* should be 'delete[]'**  
**warning** The type of an object to be `delete`'d is usually a pointer. This is because operator `new` always returns a pointer and `delete` may only delete that allocated via `new`. Perhaps this is a programmer error attempting to delete a local or global array? [16]
- 1509** **the destructor of derived class *type* is non-trivial, but the destructor of base class *type* is not virtual**  
**warning** The indicated class is a base class for some derived class. It has a non-virtual destructor. Was this a mistake? It is conventional to virtualize destructors of base classes so that it is safe to `delete` a base class pointer. [16]
- 1510** **the destructor of derived class *type* is non-trivial, but no destructor provided for base class *type***  
**warning** The indicated class is a base class for some derived class that has a destructor. The base class does not have a destructor. Is this a mistake? The difficulty that you may encounter is this; if you represent (and manipulate) a heterogeneous collection of possibly derived objects via a pointer to the base class then you will need a virtual base class destructor to invoke the derived class destructor. [17, Section 4]
- 1511** **member function *symbol* hides non-virtual 'string' member *symbol***  
**warning** The named member of a derived class hides a similarly named member of a base class. Moreover, the base class member is not `virtual`. Is this a mistake? Was the base member supposed to have been declared `virtual`? By unnecessarily using the same name, confusion could be created. The *string* parameter contains the member access of the hidden function (one of `public`, `protected`, or `private`) which can be used with `-estring` to suppress the message for e.g. hidden private functions using `-estring(1511,private)`. This message is not issued for assignment operators or deleted member functions.  
**Supports AUTOSAR17 Rule A10-2-1**  
**Supports AUTOSAR19 Rule A10-2-1**
- 1513** **storage class ignored for member declaration**  
**warning** A class member was declared with the `extern` or `register` storage class specifier. Member declarations are not allowed to be declared as `extern` or `register`.
- 1516** **data member *symbol* hides inherited member *symbol***  
**warning** A data member of a class happens to have the same name as a member of a base class. Was this deliberate? Identical names can cause confusion. To inhibit this message for a particular symbol or for an identifiable set of symbols use `-esym()`.
- 1520** **multiple *copy/move* assignment operators for class *symbol***  
**warning** More than one copy assignment operator or more than one move assignment operator has been declared for a given class. For example, for class `X` there may have been declared:

```
void operator=(X);
void operator=(X) const;
```

Which is to be used for assignment?

- 1521 multiple *copy/move* constructors for class *symbol***  
**warning** For a given class, more than one function was declared that could serve as a copy or move constructor. Typically, this means that you declared both `X( X& )` and `X( const X& )` for the same class. This is probably a mistake.
- 1524 new in constructor for class *symbol* which has no explicit destructor**  
**warning** A call to `new` has been found in a constructor for a class for which no explicit destructor has been declared. A destructor was expected because how else can the storage be freed? [14, Section `class.free`]
- 1526 undefined member function *symbol***  
**warning** A non-private member function (named in the message) of a non-library class was not defined. This message is suppressed for unit checkout (`-unit_check` option).
- 1527 undefined static data member *symbol***  
**warning** A static data member (named in the message) of a non-library class was not defined. In addition to its declaration within the class, it must be defined in some module. This message is suppressed for unit checkout (`-unit_check` option).
- 1529 assignment operator *symbol* should check for self-assignment**  
**warning** The assignment operator does not appear to be checking for assignment of the value of a variable to itself (assignment to `this`). Specifically, PC-lint Plus is looking for the first statement of the assignment operator be an if statement which compares `this` to either `&argument` or `std::addressof(argument)` using either `==` or `!=`.  
 It is important to check for a self assignment so as to know whether the old value should be subject to a delete operation. This is often overlooked by a class designer since it is counter-intuitive to assign to oneself. But, through the magic of aliasing (pointers, references, function arguments) it is possible for an unsuspecting programmer to stumble into a disguised self-assignment. [18, Item 17]
- If you are currently using the following test
- ```
if( arg == *this)
```
- we recommend you replace this with the more efficient:
- ```
if( &arg == this || arg == *this)
```
- Supports AUTOSAR17 Rule A12-8-5  
 Supports AUTOSAR19 Rule A12-8-5
- 1531 member allocation function *symbol* of non-final class *symbol* does not reference the allocation size**  
**warning** A member allocation function (operator `new` or `delete`, including array forms) was declared within a non-final class and does not appear to utilize the dynamic size of the allocation. The size parameter may have been omitted entirely (in the case of operator `delete`) or was never referenced within the function. Because the enclosing class is not final, another class could derive from this class and inherit a member allocation function that relies on a fixed size appropriate only for the base class.

- 1532** *operator-delete* should check first parameter for null  
**warning** A member `operator delete` should check its argument for NULL as it is unspecified whether deallocation functions are invoked when a null pointer is deleted.
- 1533** repeated friend declaration for symbol *symbol*  
**warning** A `friend` declaration for a particular symbol (class or function) was repeated in the same class. Usually this is a harmless redundancy.
- 1534** static variable *symbol* defined within inline function *symbol* in header '*file*'  
**warning** A static variable (*symbol*) was found within an inline function within a header file. This can be a source of error since the static variable will not retain the same value across multiple modules. Rather each module will retain its own version of the variable. If multiple modules need to use the function then have the function refer to an external variable rather than a static variable. Conversely, if only one module needs to use the function then place the definition of the function within the module that requires it. [1, Item 26]
- 1535** member function *symbol* exposes lower access pointer member *symbol*  
**warning** A member function is returning an address being held by the indicated member symbol (presumably a pointer). The member's access (such as `private` or `protected`) is lower than the access of the function returning the address.  
**Supports CWE-495** - Private Data Structure Returned From A Public Method  
**Supports CWE-767** - Access to Critical Private Variable via Public Method
- 1536** member function *symbol* exposes lower access member *symbol*  
**warning** A member function is returning the non-const address of a member either directly or via a reference. Moreover, the member's access (such as `private` or `protected`) is lower than the access of the function returning the address. For example:
- ```
class X {
private:
    int a;
public:
    int *f() { return &a; }
};
```
- This looks like a breach of the access system [6, Item 30]. You may lower the access rights of the function, raise the accessibility of the member, or make the return value a `const` pointer or reference. In the above example you could change the function to:
- ```
const int *f() { return &a; }
```
- Supports MISRA C++ Rule 9-3-1**  
**Supports MISRA C++ Rule 9-3-2**  
**Supports CWE-495** - Private Data Structure Returned From A Public Method  
**Supports CWE-767** - Access to Critical Private Variable via Public Method
- 1537** const member function *symbol* exposes pointer member *symbol* as pointer to non-const  
**warning** A `const` function is behaving suspiciously. It is returning a pointer data member (or equivalently a pointer to data that is pointed to by a data member). For example,
- ```
class X {
    int *p;
    int *f() const { return p; }
```

```
};
```

Since `f` is supposedly `const` and since `p` is presumptively pointing to data that is logically part of `class X`, we certainly have the potential for a security breach. Either return a pointer to `const` or remove the `const` modifier to the function. [6, Item 29] This message will not be issued when returning a function pointer.

Note, if a `const` function returns the address of a data member then a 605 (capability increase) is issued.

**Supports CWE-495** - *Private Data Structure Returned From A Public Method*

**1538 warning** base class *name* absent from initializer list for *copy/move* constructor

The indicated base class did not appear in the initializer list for a copy or move constructor. Was this an oversight? If the initializer list does not contain an initializer for a base class, the default constructor is used for the base class. This is not normally appropriate for a copy or move constructor. The following is more typical:

```
class B { ... };
class D : public B {
    D( const D &arg ) : B( arg ) { ... }
    ...
};
```

**1539 warning** member *symbol* not assigned by assignment operator *symbol*

The indicated *symbol* was not assigned by an assignment operator. Was this an oversight? It is not strictly necessary to initialize all members in an assignment operator because the '`this`' class is presumably already initialized. But it is easy to overlook the assignment of individual members. It is also easy to overlook your responsibility to assign base class members. This is not done for you automatically. [6, Item 16]

The message is not given for `const` members or reference members. If you have a member that is deliberately not initialized you may suppress the message for that member only using `-esym`.

**Supports CWE-456** - *Missing Initialization of a Variable*

**1540 warning** non-static pointer data member *symbol* not deallocated nor zeroed by destructor *symbol*

The indicated member is a non-static pointer member of a class that was apparently not freed by the class destructor. Was this an oversight? By freeing, we mean either a call to the `free()` function or use of the `delete` operator. If the pointer is intended only to point to static information during its lifetime then, of course, it never should be freed. In that case you should signal closure by assigning it the NULL pointer (0).

**1541 warning** non-static data member *symbol* possibly not initialized by constructor *symbol*

The indicated non-static data member may not have been initialized by the specified constructor. Specifically, this means that the member does not have an in-class initializer, was not present in the member-initializer list, and was assigned a value (directly or indirectly by a called function) in only some of the paths that the constructor takes.

**1544 warning** value of variable *symbol* is indeterminate due to run time initialization of *symbol*

A variable (identified by *symbol*) was used in the run-time initialization of a static variable. However this variable itself was initialized at run-time. Since the order of initialization cannot be predicted this is the source of a possible error.

Whereas addresses are completely known at initialization time, values may not be. Whether the value or merely the address of a variable is used in the initialization of a second variable is not an easy thing to

determine when an argument is passed by reference or via pointer. For example, given `struct X` defined as:

```
struct X {
    X(const X &);
    X();
};
```

a module containing the definition:

```
extern X x1;
X x2 = x1;
```

and another module that defines `x1`, PC-lint Plus will issue a message similar to:

```
warning 1544: value of variable 'x2' is indeterminate due to run
             time initialization of 'x1'
X x2 = x1;
^
```

It is theoretically possible, but unlikely, that the constructor `X()` is interested only in the address of its argument and not its current value. If so, it only means you will be getting a spurious report, which you can suppress based on variable name. However, if the `const` is missing when passing a reference parameter (or a pointer parameter) then we cannot easily assume that values are being used. In this case no report will be issued. The moral is that if you want to get the checking implied by this message you should make your constructor reference arguments `const`. This message is suppressed for unit checkout (`-unit_check` option).

**1546 throw outside try in destructor body**  
**warning** The body of a destructor (signature provided within the message) contains a `throw` not within a `try` block. This is dangerous because destructors are themselves triggered by exceptions in sometimes unpredictable ways. The result can be a perpetual loop. [1, Item 11]  
**Supports MISRA C++ Rule 15-3-1**  
**Supports MISRA C++ Rule 15-5-1**  
**Supports MISRA C++ Rule 15-5-3**

**1547 assignment of array of derived class to pointer to base class (*context*)**  
**warning** An assignment from an array of a derived class to a pointer to a base class was detected. For example:

```
class B { };
class D : public B {};
D a[10];
B *p = a;           // Warning 1547
B *q = &a [0];      //OK
```

In this example `p` is being assigned the address of the first element of an array. This is fraught with danger since access to any element other than the zeroeth must be considered an error (we presume that `B` and `D` actually have or have the potential to have different sizes). [1, Item 3]

We do not warn about the assignment to `q` because it appears that the programmer realizes the situation and wishes to confine `q` to the base object of the zeroeth element of `a` only. As a further precaution against inappropriate array access, out of bounds warnings are issued for subsequent references to `p[1]` and `q[1]`.

**1548 exception specification in declaration does not match previous declaration**  
**warning** The exception specification of a function begins with the keyword `'throw'` and follows the prototype. Two declarations were found for the same function with inconsistent exception specifications.  
**Supports MISRA C++ Rule 15-4-1**



- 1549 warning** **exception of type *type* thrown from function *symbol* is not in throw list**  
 An exception was thrown (i.e., a `throw` was detected) within a function and not within a `try` block; more over the function contains a `throw` specification but the exception thrown was not on the list. If you provide an exception specification, include all the exception types you potentially will throw. [19, Item 14]  
**Supports MISRA C++ Rule 15-5-2**
- 1550 warning** **exception '*name*' thrown by function *symbol* is not on throw-list of function *symbol***  
 A function was called (first *symbol*) that was declared as throwing an exception. The call was not made from within a `try` block and the function making the call (second *symbol*) had an exception specification that did not include one of the types specified in the called function's exception specification. Either add the exception to the calling function's exception list, or place the call inside a `try` block and catch the `throw`. [1, Item 14]
- 1551 warning** **function *symbol* called outside of a try block in destructor *symbol* is not declared as never throwing**  
 A call to a function (given by the first *symbol*) was made from within a destructor given by the second *symbol*. The called function was declared as potentially throwing an exception. Such exceptions need to be caught within a `try` block because destructors should never throw exceptions. [1, Item 11].
- 1552 warning** **converting pointer to array of derived to pointer to base**  
 This warning is similar to Warning 1547 and is sometimes given in conjunction with it. It uses value tracking to determine that an array (that could be dynamically allocated) is being assigned to a base class pointer. For example,
- ```
Derived *d = new Derived[10];
Base *b;
b = d;      // Warning 1552
b = &d[0];  //OK
```
- This case is an issue because if one tries to access `b[i]`, where `i` is an index value, the compiler will attempt to access the object with the address `i * sizeof(Base)` from `b`. However, since the size of `Derived` is almost certain to be larger than the size of `Base`, this object is not the `i`-th `Derived` object.
- [1, Item 3] Also, see the article by Mark Nelson (Bug++ of the Month, Windows Developer's Journal, May 1997, pp. 43-44).
- 1554 warning** **shallow pointer copy of *symbol* in copy constructor *symbol***  
 In a copy constructor a pointer was merely copied rather than recreated with new storage. This can create a situation where two objects have the same data and this, in turn, causes problems when these objects are deleted or modified. For example, the following class will draw this warning:
- ```
class X {
    char *p;
    X(const X &x) { p = x.p; }
};
```
- Here, member `p` is expected to be recreated using `new` or some variant.
- 1555 warning** **shallow pointer copy of *symbol* in copy assignment operator *symbol***  
 In a copy assignment operator a pointer was merely copied rather than recreated with new storage. This

can create a situation where two objects have the same data and this, in turn, causes problems when these objects are deleted or modified. For example, the following class will draw this warning:

```
class X {
    char *p;
    X &operator=(const X &x) { p = x.p; }
};
```

Here, member `p` is expected to be recreated using `new` or some variant.

#### 1556 initialization could be confused with array allocation

**warning** A `new` expression had the form `new T(integer)` where type `T` has no constructor. For example:

```
new int(10);
```

will draw this warning. The expression allocates an area of storage large enough to hold one integer. It then initializes that integer to the value 10. Could this have been a botched attempt to allocate an array of 10 integers? Even if it was a deliberate attempt to allocate and initialize a single integer, a casual inspection of the code could easily lead a reader astray.

The warning is only given when the type `T` has no constructor. If `T` has a constructor then either a syntactic error will result because no constructor matches the argument or a match will be found. In the latter case no warning will or should be issued.

#### 1558 inline virtual function is unusual

**warning** The function declared both `virtual` and `inline` has been detected. An example of such a situation is as follows:

```
class C {
    virtual inline void f();
};
```

Virtual functions by their nature require an address and so inlining such a function seems contradictory. We recommend that the `inline` function specifier be removed.

#### 1559 uncaught exception '*name*' may be thrown in destructor *symbol*

**warning** The named exception occurred within a `try` block and was either not caught by any handler or was caught but then thrown from the handler. Destructors should normally not throw exceptions. [1, Item 11]

**Supports CWE-248 - Uncaught Exception**

#### 1560 uncaught exception '*name*' not on throw-list of function *symbol*

**warning** A direct or indirect `throw` of the named exception occurred within a `try` block and was either not caught by any handler or was rethrown by the handler. Moreover, the function has an exception specification and the uncaught exception is not on the list. Note that a function that fails to declare a list of thrown exceptions is assumed to potentially throw any exception.

**Supports MISRA C++ Rule 15-3-4**

**Supports CWE-248 - Uncaught Exception**

#### 1562 exception specification for *symbol* is not a subset of *symbol*

**warning** The first *symbol* is that of an overriding virtual function for the second *symbol*. The exception specification for the first was found not to be a subset of the second. For example, it may be reasonable to have:

```
struct B { virtual void f() throw(B); };
struct D:B { virtual void f() throw(D); };
```

Here, although the exception specifications are not identical, the exception D is considered a subset of the base class B.

It would not be reasonable for `D::f()` to throw an exception outside the range of those thrown by `B::f()` because in general the compiler will only see calls to `B::f()` and it should be possible for the compiler to deduce what exceptions could be thrown by examining the static call.

### 1563 unparenthesized assignment as false expression of conditional operator

**warning** The third argument to `?:` contained an unparenthesized assignment operator such as

```
p ? a : b = 1
```

If this is what was intended you should parenthesize the third argument as in:

```
p ? a : (b = 1)
```

Not only is the original form difficult to read but C, as opposed to C++, would parse this as:

```
(p ? a : b) = 1
```

### 1564 converting integer constant expression, which evaluates to *integer* but is not an integer literal equal to zero or one, to bool

**warning**

An integer constant expression other than the integer literal 0 or 1 was implicitly converted to `bool`. It may have been unintended that the value will essentially be discarded as the result of the expression will depend only on whether or not it was zero. If the Boolean use of this expression was intentional it could be explicitly compared `!= 0` to clarify the intent (note that Boolean comparison against integral values other than 0 is discouraged because all non-zero values are equally true — see message [697](#)).

### 1565 non-static data member *symbol* not initialized by initializer function *symbol*

**warning**

A function dubbed 'initializer' by a `-sem` option is not initializing (i.e., assigning to) every data member of a class. `const` members theoretically can be initialized only via the constructor so that these members are not candidates for this message.

### 1566 non-static data member *symbol* may have been initialized in a separate method but no `'-sem(name,initializer)'` option was seen

**warning**

A class data member (whose name and location are indicated in the message) was not directly initialized by a constructor. It may have been initialized by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint Plus that the separately called function is in fact an 'initializer'. For example:

```
class A {
    int a;
public:
    void f();
    A() { f(); }
};
```

Here `f()` is presumably serving as an initializer for the constructor `A::A()`. To inform PC-lint Plus of this situation, use the option:

```
-sem( A::f, initializer )
```

This will suppress Warning 1566 for any constructor of `class A` that calls `A::f`.

- 1570 binding reference field *symbol* to non-reference parameter *symbol***  
**warning** In a constructor initializer, a reference class member is being initialized to bind to an auto variable. Consider:

```
class X { int &n; X(int k) :n(k) {} };
```

In this example member `n` is being bound to variable `k`, which although a parameter, is nonetheless placed into `auto` storage. But the lifetime of `k` is only the duration of the call to the constructor, whereas the lifetime of `n` is the lifetime of the class object constructed.

- 1571 returning an auto variable *symbol* via a reference type**  
**warning** A function that is declared to return a reference is returning an `auto` variable (that is not itself a reference). The `auto` variable is not guaranteed to exist beyond the lifetime of the function. This can result in unreliable and unpredictable behavior.

- 1572 initializing a static reference variable with an auto variable *symbol***  
**warning** A `static` variable has a lifetime that will exceed that of the `auto` variable that it has been bound to. Consider

```
void f( int n ) { static int& r = n; ... }
```

The reference `r` will be permanently bound to an `auto` variable `n`. The lifetime of `n` will not extend beyond the life of the function. On the second and subsequent calls to function `f` the static variable `r` will be bound to a non-existent entity.

- 1576 explicit specialization is not in the same file as specialized function template *symbol***  
**warning** An explicit specialization of a function template was found to be declared in a file other than the one in which the corresponding function template is declared. Two identical calls in two different modules on the same function template could then have two differing interpretations based on the inclusion of header files. The result is undefined behavior.

As if this wasn't enough, if the explicit specialization could match two separate function templates then the result you obtain could depend upon which function templates are in scope.

See also the next message.

**Supports AUTOSAR17 Rule M14-7-3**

**Supports AUTOSAR19 Rule A14-7-2**

**Supports MISRA C++ Rule 14-7-3**

- 1578 non-static pointer data member *symbol* not deallocated nor zeroed by cleanup function *symbol***  
**warning** The indicated member is a non-static data member of a class that was apparently not cleared by a function that had previously been given the `cleanup` semantic. By clearing we mean that the pointer was either zeroed or the storage associated with the pointer released via the `free` function or its semantic equivalent or some form of `delete`. See also Warning [1540](#).

- 1579 non-static pointer data member *symbol* may have been zeroed or freed in a separate method but no '-sem(name,cleanup)' option was seen**  
**warning**

A class data member (whose name and location are indicated in the message) was not directly freed by the class destructor. There was a chance that it was cleared by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint Plus that the separately called function is in fact a '`cleanup`' function. For example:

```
class A {
    int *p;
```

```
public:
    void release_ptrs();
    ~A() { release_ptrs(); }
};
```

Here `release_ptrs()` is presumably serving as a `cleanup` function for the destructor `~A::A()`. To inform PC-lint Plus of this situation, use the option:

```
-sem( A::release_ptrs, cleanup )
```

A separate message (Warning [1578](#)) will be issued if the `cleanup` function fails to clear all pointers. See also Warning [1566](#).

**1705** *static member **symbol** could be accessed using a nested name specifier instead of applying operator '**string**' to an instance of class **symbol***

info

A static class member was accessed using a class object and `->` or `.` notation. For example:

```
s.member
```

or

```
p->member
```

But an instance of the object is not necessary. It could just as easily have been referenced as:

```
X::member
```

where `X` is the class name. [14, Section `class.static`]

**1706** *extra qualification on member **symbol** within a class*

info

Class members within a class are not normally declared with the scope operator. For example:

```
class X { int X::n; };
```

will elicit this message. If the (redundant) class specification (`X::`) were replaced by some different class specification and the declaration was not `friend` an error ([1040](#)) would be issued. [13, Section 9.2]

**1707** *static assumed for **symbol***

info

operator `new()` and operator `delete()`, when declared as member functions, should be declared as `static`. They do not operate on an object instantiation (implied `this` pointer). [13, Section 12.5]

**1710** *missing '**typename**' prior to dependent type name '**string**'*

info

This message is issued when the standard requires the use of '`typename`' to disambiguate the syntax within a template where it may not be clear that a name is the name of a type or some non-type. (See C++ Standard [14], Section `temp.res`, Para 2). Consider:

```
template <class T>
class A {
    T::N x;    // Info 1710
};
```

while technically ill-formed, some compilers will accept this construct since the only interpretation consistent with valid syntax is that `T::N` represents a type. (But if the '`x`' weren't there it would be taken as an access declaration and more frequently would be a non-type.)

**1711 class *symbol* has a virtual function but is not inherited**

**info** The given class has a virtual function but is not the base class of any derivation. Was this a mistake? There is no advantage to making member functions virtual unless their class is the base of a derivation tree. In fact, there is a disadvantage because there is a time and space penalty for virtual functions. This message is not given for library classes and is suppressed for unit checkout. [17, section 4]

**1714 member function *symbol* not referenced**

**info** A non-private member function was not referenced. This message is automatically suppressed for unit checkout (`-unit_check`) and for members of a library class.

**Supports AUTOSAR17 Rule M0-1-10**

**Supports AUTOSAR19 Rule M0-1-10**

**Supports MISRA C++ Rule 0-1-10**

**1715 static member *symbol* not referenced**

**info** A static data member of a class was not referenced. This message is automatically suppressed for unit checkout (`-unit_check`) and for members of a library class.

**Supports AUTOSAR17 Rule M0-1-3**

**Supports AUTOSAR19 Rule M0-1-3**

**Supports MISRA C++ Rule 0-1-3**

**1719 reference parameter of copy assignment operator *symbol* should be *type***

**info** The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not a reference then your program is subject to implicit function calls and less efficient operation. [13, Section 13.4.3]

**1720 reference parameter of copy assignment operator *symbol* should be *type***

**info** The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not `const` then your program will not be diagnosed as completely as it might otherwise be. [13, Section 13.4.3]

**1721 operator=() for class *symbol* is not a copy nor move assignment operator**

**info** Class assignment operators typically have one of the following forms:

```
X& operator=(const X &); // copy assignment
X& operator=(X &&);      // move assignment
```

A member function whose name is `operator=` but does not have one of these forms is unusual and may be a subtle source of bugs. If this is not an error you may selectively suppress this message for the given class.

**1722 assignment operator *symbol* should return *type***

**info** The typical assignment operator for a class X is of the form:

```
X& operator =(const X &);
```

The reason for returning a reference to class is to support multiple assignment as in:

```
a = b = c
```

See also messages [9409](#) and [9412](#).

[13, Section 13.4.3]

**1724 parameter of copy constructor for class *symbol* should be a const reference**

**info** The parameter for a copy constructor is generally declared as a reference to `const`. This signature is not only standard practice but is also the way a compiler-provided copy constructor is generated. Using a reference that is not to `const` will prevent it from accepting rvalues, including temporary objects (although an applicable move constructor would take precedence if available). The omission of `const` can also cause unexpected results when a copy constructor is declared as deleted, for example:

```
struct A {
    A(int) { }
    operator bool() { }
    A(A&) = delete; // should be const A&
};
void f() {
    A b = A(5); // compiles even though the copy constructor is deleted
}
```

If the copy constructor in this example had been declared to take a reference to `const`, the temporary would have triggered the selection of the copy constructor and led to the compilation error that would probably be expected when attempting an operation resembling copy construction. Without `const`, the compiler does not consider the copy constructor because the parameter type cannot accept an xvalue, and instead a circuitous and perhaps unexpected conversion sequence is chosen.

**1726 deletion of pointer to const function parameter *symbol***

**info** The delete operator was applied to either a pointer to const or an array of const. While permitted by the C++ standard, the practice is questioned. If a function didn't have the capability of writing into the area designated by a pointer why would we suppose it to be ok to delete the area?

**1727 function *symbol* declared inline here was not previously declared inline**

**info** A function declared or defined inline was not previously declared inline. Was this intended? If this is your standard practice then suppress this message. [13, Section 9.3.2]

**1728 function *symbol* was previously declared inline**

**info** A function was previously declared or defined inline. The inline modifier is absent from the current declaration or definition. Was this intended? If this is your standard practice then suppress this message. [13, Section 9.3.2]

**1729 initializer inversion: *field/base class symbol* appears before, but will be initialized after, *field/base symbol* in member initializer list**

In a constructor initializer the order of evaluation is determined by the member order not the order in which the initializers are given. At least one of the initializers was given out of order. Was there a reason for this? Did the programmer think that by changing the order that he/she would affect the order of evaluation? Place the initializers in the order of their occurrence within the class so that there can be no mistaken assumptions. [6, Item 13]

**Supports AUTOSAR17 Rule A8-5-1**

**Supports AUTOSAR19 Rule A8-5-1**

**1730 *string* *string* type was previously declared as a *string* *string***

**info** An object is declared both with the keyword `class` and with the keyword `struct`. Though this is legal it is suspect. [13, Section 7.1.6]

**1731 public virtual function *symbol***

**info** A class member function was declared both `public` and `virtual`. Some authors, see [11, Rule 39], have advocated avoiding public virtual functions because such functions are both part of the public interface and a customization point, aspects often with conflicting motives and audiences. Rather than make the virtual function public consider making it protected. This way members of the hierarchy may still customize behavior.

**1732 new in constructor for class *symbol* which has no user-provided copy assignment operator**

**info** Within a constructor for the cited class, there appeared a `new`. However, no assignment operator was declared for this class. Presumably some class member (or members) points to dynamically allocated memory. Such memory is not treated properly by the default assignment operator. Normally a custom assignment operator would be needed. Thus, if `x` and `y` are both of type *symbol*

```
x = y;
```

will result in pointer duplication. A later `delete` would create chaos. [6, Item 11]

**1733 new in constructor for class *symbol* which has no user-provided copy constructor**

**info** Within a constructor for the cited class, there appeared a `new`. However, no copy constructor was declared for this class. Presumably, because of the `new`, some class member (or members) points to dynamically allocated memory. Such memory is not treated properly by the default copy constructor. Normally a custom copy constructor would be needed. [6, Item 11]

**1735 parameter '*string*' of virtual function *symbol* has a default argument**

**info** A virtual function was detected with a default parameter. For example:

```
class B {
    virtual void f( int n = 5 );
    ...
};
```

The difficulty is that every virtual function `f` overriding this virtual function must contain a default parameter and its default parameter must be identical to that shown above. If this is not done, no warnings are issued but behavior may have surprising effects. This is because when `f()` is called through a base class pointer (or reference) the function is determined from the actual type (the dynamic type) and the default argument is determined from the nominal type (the static type). [6, Item 38].

**Supports AUTOSAR17 Rule M8-3-1**

**Supports AUTOSAR19 Rule M8-3-1**

**Supports MISRA C++ Rule 8-3-1**

**1736 redundant access specifier (*string*)**

**info** An access specifier (one of `public`, `private`, or `protected` as shown in *string*) is redundant. That is, the explicitly given access specifier did not have to be given because an earlier access specifier of the same type is currently active. This message is NOT given for an access specifier that is the first item to appear in a class definition. Thus,

```
class abc { private: ...
```



does not draw this message. The reason this message is issued is because it is very easy to make the following mistake.

```
class A {
public:
    ...
public:
    ...
}
```

In general there are no compiler warnings that would result from such an unintentional botch.

**1738** *copy/move constructor for class **symbol** explicitly invokes non-copy/move constructor for base class **symbol***

In an initializer list for a copy constructor, a base class constructor was invoked. However, this base class constructor was not itself a copy constructor. We expect that copy constructors will invoke copy constructors. Was this an oversight or was there some good reason for choosing a different kind of constructor? If this was deliberate, suppress this message. See also message [1538](#).

**1746** *non-reference parameter **symbol** of function **symbol** could be reference to const*

**info** The indicated parameter is a candidate to be declared as a `const` reference. For example:

```
void f( X x ) {
    // x not modified.
}
```

Then the function definition can be replaced with:

```
void f( const X &x ) {
    // x not modified.
}
```

The result may be more efficient since less information needs to be placed onto the stack and a constructor need not be called.

The message is only given with class-like arguments (including `structs` and `unions`) and only if the parameter is not subsequently modified or potentially modified by the function. The parameter is potentially modified if it is passed to a function whose corresponding parameter is a reference (not `const`) or if its address is passed to a non-`const` pointer. [6, Item 22].

This message is not issued for `extern "C"` functions, which presumably cannot employ references.

**1747** *binary operator **symbol** returns reference type **type***

**info** An operator-like function was found to be returning a reference. For example:

```
X &operator+ ( X &, X & );
```

This is almost always a bad idea. [6, Item 23]. You normally can't return a reference unless you allocate the object, but then who is going to delete it. The usual way this is declared is:

```
X operator+ ( X &, X & );
```

**1748** *non-virtual base class **symbol** included twice in class **symbol***

**info** Through indirect means, a given class was included at least twice as a base class for another class. At least one of these is not virtual. Although legal, this may be an oversight. Such base classes are usually marked

virtual resulting in one rather than two separate instances of the base class. This is done for two reasons. First, it saves memory; second, references to members of such a base class will not be ambiguous.

Supports AUTOSAR17 Rule M10-1-3

Supports AUTOSAR19 Rule M10-1-3

Supports MISRA C++ Rule 10-1-3

#### 1749 base class *symbol* of class *symbol* need not be virtual

**info** The designated base class is a direct base class of the second class and the derivation was specified as 'virtual'. But the base class was not doubly included (using this link) within any class in the entire project. Since a virtual link is less efficient than a normal link this may well be an unenlightened use of 'virtual'. [1, Item 24]. The message is inhibited if unit checkout (`-unit_check`) is selected.

#### 1751 anonymous namespace declared in a header file

**info** An unnamed namespace was used in a header file.

Supports AUTOSAR17 Rule M7-3-3

Supports AUTOSAR19 Rule M7-3-3

Supports MISRA C++ Rule 7-3-3

#### 1752 catch parameter is not a reference

**info** This message is issued for every `catch` parameter that is not a reference and is not numeric. The problem with pointers is a problem of ownership and delete responsibilities; the problem with a non-ref object is the problem of slicing away derivedness [1, Item 23].

Supports AUTOSAR17 Rule A15-3-5

Supports AUTOSAR19 Rule A15-3-5

Supports MISRA C++ Rule 15-3-5

#### 1753 overloading operator *symbol* precludes short-circuit evaluation

**info** This message is issued whenever an attempt is made to declare one of these operators as having some user-defined meaning:

```
operator ||
operator &&
operator ,
```

The difficulty is that the working semantics of the overloaded operator is bound to be sufficiently different from the built-in operators, as to result in possible confusion on the part of the programmer. With the built-in versions of these operators, evaluation is strictly left-to-right. With the overloaded versions, this is not guaranteed. More critically, with the built-in versions of `&&` and `||`, evaluation of the 2nd argument is conditional upon the result of the first. This will never be true of the overloaded version. [1, Item 7].

Supports AUTOSAR17 Rule M5-2-11

Supports AUTOSAR19 Rule M5-2-11

Supports MISRA C++ Rule 5-2-11

#### 1754 expected symbol '*string*' to be declared for class *symbol*

**info** The first *symbol* is of the form: `operator op=` where *op* is a binary operator. A binary operator *op* was declared for type *X* where *X* is identified by the second *symbol*. For example, the appearance of:

```
X operator+( const X &, const X & );
```

somewhere in the program would suggest that a `+=` version appear as a member function of class *X*. This is not only to fulfill reasonable expectations on the part of the programmer but also because `operator+=` is likely to

be more efficient than `operator+` and because `operator+` can be written in terms of `operator+=`. [1, Item 22]

The message is also given for member binary operators. In all cases the message is not given unless the return value matches the first argument (this is the implicit argument in the case of a member function).

**1756 variable *symbol* has '*static/thread*' storage duration and non-POD type *type***

**info** A variable with either *static* or *thread* storage duration (the string parameter indicates which) was declared with a non-POD type. C++ doesn't define the order in which static or thread storage duration variables in different modules are initialized which can be a source of subtle errors. This message isn't issued for `constexpr` variables.

**Supports AUTOSAR17 Rule A3-3-2**

**1757 discarded instance of member post-*string* operator**

**info** A postfix increment or postfix decrement operator was used in a context in which the result of the operation was discarded. For example:

```
X a;
...
a++;
```

In such contexts it is just as correct to use prefix decrement/increment. For example this could be replaced with:

```
X a;
...
++a;
```

The prefix form is (or should be) more efficient than the postfix form because, in the case of user-defined types, it should return a reference rather than a value (see [1758](#) and [1759](#)). This presumes that the side effects of the postfix form are equivalent to those of the prefix form. If this is not the case then either make them equivalent (the preferred choice) or turn this message off. See also [2902](#), which is issued for non-class types. [1, Item 6].

**1758 prefix *symbol* does not return a reference**

**info** To conform with most programming expectations, a prefix increment/decrement operator should return a reference. Returning a reference is both more flexible and more efficient [1, Item 6].

The expected form is as shown below:

```
class X {
    X & operator++();
    X operator++( int );
    ...
};
```

**1759 postfix *symbol* returns a reference**

**info** To conform with most programming expectations, a postfix increment/decrement operator should return a value as opposed to a reference. [1, Item 6]. See example in message [1758](#).

**1762 member function *symbol* could be made const**

**info** The indicated (non-static) member function did not modify member data and did not call non-const functions.

Moreover, it does not make any deep modification to the class member. A modification is considered deep if it modifies information indirectly through a class member pointer. Therefore, it could and probably should be declared as a `const` member function.

**Supports AUTOSAR17 Rule M9-3-3**

**Supports AUTOSAR19 Rule M9-3-3**

**Supports MISRA C++ Rule 9-3-3**

### 1763 **const member function *symbol* contains deep modification**

**info** The designated symbol is a member function declared as `const`. Though technically valid, the `const` may be misleading because the member function modifies (or exposes) information indirectly referenced by the object. For example:

```
class X {
    char *pc;
    char &get(int i) const { return pc[i]; }
};
```

results in Info 1763 for function `X::get`. This is because the function exposes information indirectly held by the class `X`.

Experts [19] recommend that a pair of functions be made available in this situation:

```
class X {
    char *pc;
    const char & get(int i) const { return pc[i]; }
    char & get(int i) { return pc[i]; }
};
```

In this way, if the object is `const` then only the `const` function will be called, which will return the protected reference. Related messages are also [1762](#) and [1962](#). See also [6, Item 29] for a further description.

### 1764 **reference parameter *symbol* of function *symbol* could be reference to const**

**info** As an example:

```
int f( int & k ) { return k; }
```

can be redeclared as:

```
int f( const int & k ) { return k; }
```

Declaring a parameter a reference to `const` offers advantages that a mere reference does not. In particular, you can pass constants, temporaries and `const` types into such a parameter where otherwise you may not. In addition it can offer better documentation.

Other situations in which a `const` can be added to a declaration are covered in messages [818](#), [952](#), [953](#) and [954](#).

**Supports AUTOSAR17 Rule M7-1-2**

**Supports AUTOSAR19 Rule M7-1-2**

**Supports MISRA C++ Rule 7-1-2**

### 1766 **catch(...) encountered without preceding catch clause**

**info** An ellipsis was used in a catch handler resulting in a handler that will catch any exception. This "catch-all" handler was not preceded by one or more catch handlers in the same try block meaning that this handler will be responsible for processing all exceptions. Catch-all exception handlers are generally considered a bad practice due to the inability to distinguish between different types of exceptions and the potential to hide serious issues. The somewhat less serious use of an exception handler with preceding catch clauses is

diagnosed by message [1966](#).

**Supports** CWE-396 - *Declaration of Catch for Generic Exception*

**1768** *access virtual function **symbol** overrides **access** function in base class **symbol***

**info** An overriding virtual function has an access (public, protected or private) in the derived class different from the access of the overridden virtual function in the base class. Was this an oversight? Since calls to the overriding virtual function are usually made through the base class, making the access different is unusual (though legal).

**1771** *function **symbol** replaces global function*

**info** This message is given for `operator new` and `operator delete` (and for their [] cousins) when a definition for one of these functions is found. Redefining the built-in version of these functions is not considered sound programming practice. [1, Item 23]

**1772** *assignment operator **symbol** should return **\*this***

**info** The assignment operator should return `*this`. This is to allow for multiple assignments as in:

```
a = b = c;
```

It is also better to return the object that has just been modified rather than the argument. [6, Item 15]

**1773** *casting away **const/volatile** qualifier without **const\_cast** (**type** to **type**)*

**info** An attempt was made to cast away `const`. This can break the integrity of the `const` system. This message will be suppressed if you use `const_cast`. Thus:

```
char *f(const char *p) {
    if (test())
        return (char *)p;           // Info 1773
    else
        return const_cast<char *>(p); // OK
}
```

See [6, Item 21].

**1774** *only **dynamic\_cast** can indicate a failure by returning null – cast from **type** to **type** will not be checked at runtime*

The result of a cast between two pointer types, other than a `dynamic_cast`, was tested for null. Only `dynamic_cast` can indicate failure with a null result. The failure of other casts at runtime produces undefined or implementation-defined behavior.

If the objective was simply to determine if the original pointer was null before accessing it through a different type then this could be clarified by testing if the original pointer is null before casting it. Casting a null pointer from one pointer type to another will generally produce a valid null pointer in the destination type, but testing this result for null may mislead the reader that this will, e.g., catch conversions from a pointer to base class to a pointer to a derived class that does not match the actual type of the object like a `dynamic_cast`.

While this message is not restricted to casts involving class types, it is only issued in C++ modules as there is no expectation in C that any cast would ever perform the runtime checking associated with `dynamic_cast`.

**Supports** AUTOSAR17 Rule M5-2-2

**Supports** AUTOSAR19 Rule M5-2-2

**Supports** MISRA C++ Rule 5-2-2

**1775 catch block does not catch any declared exceptions**

**info** A catch handler does not seem to catch any exceptions. For example:

```
try { f(); }
catch( B& ) {}
catch( D& ) {}          // Info 1775
catch( ... ) {}
catch( char * ) {}      // Info 1775
```

If `f()` is declared to throw type `D`, and if `B` is a public base class of `D`, then the first catch handler will process that exception and the second handler will never be used. The fourth handler will also not be used since the third handler will catch all exceptions not caught by the first two.

If `f()` is declared to not throw an exception then Info 1775 will be issued for all four catch handlers.

**Supports AUTOSAR17 Rule M15-3-6**

**Supports AUTOSAR19 Rule M15-3-6**

**Supports MISRA C++ Rule 15-3-6**

**1776 converting string literal to *type* is not const safe**

**info** A string literal, according to Standard C++ is typed an array of `const char`. This message is issued when such a literal is assigned to a non-const pointer. For example:

```
char *p = "string";
```

will trigger this message. This pointer could then be used to modify the string literal and that could produce some very strange behavior.

Such an assignment is legal but "deprecated" by the C++ Standard. The reason for not ruling it illegal is that numerous existing functions have their arguments typed as `char *` and this would break working code.

Note that this message is only given for string literals. If an expression is typed as pointer to `const char` in some way other than via string literal, then an assignment of that pointer to a non-const pointer will receive a more severe warning.

**Supports AUTOSAR19 Rule A2-13-4**

**Supports CERT C STR05-C - Use pointers to const when referring to string literals**

**Supports CERT C STR30-C - Do not attempt to modify string literals**

**Supports MISRA C 2012 Rule 7.4**

**1777 template recursion limit (*integer*) reached, use `-tr_limit(n)`**

**info** It is possible to write a recursive template that will contain a recursive invocation without an escape clause. For example:

```
template <class T> class A { A< A > x; };
A<int> a;
```

This will result in attempts to instantiate:

```
A<int>
A<A<int>>
A<A<A<int>>>
...
```

Using the `-vt` option (turning on template verbosity) you will see the sequence in action. Accordingly, we have devised a scheme to break the recursion when an arbitrary depth of recursion has been reached (at this writing 75). This depth is reported in the message. As the message suggests, this limit can be adjusted so

that it equals some other value.

When recursion is broken, a complete type is not used in the definition of the last specialization in the list but processing goes on.

#### 1778 assignment of string literal to variable *symbol* is not const safe

**info** This message is issued when a string literal is assigned to a variable whose type is a non-const pointer. For example:

```
char *p;  p = "abc";
```

The message is issued automatically (i.e. by default) for C++. For C, to obtain the message, you need to enable the Strings-are-Const flag (**+fsc**). This message is similar to message [1776](#) except that it is issued whenever a string constant is being assigned to a named destination.

**Supports MISRA C 2012 Rule 7.4**

#### 1779 virtual function *symbol* introduced in class *symbol* which is marked as 'final'

**info** A virtual function was declared in a class marked as **final** but does not override any base class virtual function. Since the class is final it cannot be a base class and the virtual function cannot be overridden in a derived class. If the intention is for this function to override a virtual function in a base class, mark the base class function as **virtual**. Otherwise either remove the **final** specifier on the class or the **virtual** specifier on the function.

**Supports AUTOSAR17 Rule A10-3-3**

**Supports AUTOSAR19 Rule A10-3-3**

#### 1780 returning address of reference to a const parameter *symbol*

**info** The address of a parameter that has been declared as being a reference to a **const** is being returned from a function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
const int *f( const int & n )
{ return &n; }
int g();
const int *p = f( g() );
```

Here, **p** points to a temporary value whose duration is not guaranteed. If the reference is not **const** then you will get Elective Note [1940](#).

This is an example of the Linton Convention as described by Murray [20].

**Supports MISRA C++ Rule 7-5-3**

#### 1781 passing address of const reference parameter *symbol* into caller address space

**info** The address of a parameter that has been declared as being a reference to a **const** is being assigned to a place outside the function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
void f( const int & n, const int **pp )
{ *pp = &n; }
int g();
const int *p;
... f( g(), &p );
```

Here, `p` will be made to point to a temporary value whose duration is not guaranteed. If the reference is not `const` then you will get Elective Note [1940](#).

This is an example of the Linton Convention as described by Murray [20].

#### 1782 assigning address of `const` reference parameter *symbol* to a static variable

**info** The address of a parameter that has been declared as being a reference to a `const` is being assigned to a static variable. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
const int *p;
void f( const int & n )
    { p = &n; }
int g();
... f( g() );
```

Here, `p` will be made to point to a temporary value whose duration is not guaranteed. If the reference is not `const` then you will get Elective Note [1940](#).

This is an example of the Linton Convention as described by Murray [20].

#### 1784 symbol *symbol* previously declared as "C"

**info** A *symbol* is being redeclared in the same module. Whereas earlier it had been declared with an `extern "C"` linkage, in the cited declaration no such linkage appears. E.g.

```
extern "C" void f(int);
void f(int);           // Info 1784
```

In this case the `extern "C"` prevails and hence this inconsistency probably represents a benign redeclaration. Check to determine which linkage is most appropriate and amend or remove the declaration in error.

#### 1785 implicit conversion from Boolean (*context*) (*type to type*)

**info** A Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of some other type. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

#### 1786 implicit conversion to Boolean (*context*) (*type to type*)

**info** A non-Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of type Boolean. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

#### 1787 access declarations are deprecated; use using declarations instead

**info** The C++ Standard ([14] section 7.3.3) specifically deprecates the use of access declarations. The preferred syntax is the using declaration. For example:

```
class D : public B {
    B::a;           // message 1787
    using B::a;     // preferred form and no message
};
```

In C++11, support for access declarations were removed completely. In C++11 and later modes, this message is replaced with an error.



**1788 variable *symbol* of type *symbol* is referenced only by its constructor/destructor**

**info** A variable has not been referenced other than by the constructor that formed its initial value or by its destructor or both. The location of the symbol and also its type is given in the message. For example:

```
struct A {
    A();
};
void f() { A a; }
```

will produce a 1788 for variable 'a' and for type 'A'.

It very well may be that this is exactly what the programmer wants to do, in which case you may suppress this message for this variable using the option `-esym(1788,a)`. It may also be that the normal use of `class A` is to employ it in this fashion. That is, to obtain the effects of construction and, possibly, destruction but have no other reference to the variable. In this case the option of choice would be `-esym(1788,A)`.

**1789 constructor template *symbol* cannot be a copy constructor**

**info** This message is issued for classes for which a copy constructor was not defined but a template constructor was defined. For example:

```
struct A {
    template <typename T>
    A(const T&);           // Info 1789
};
```

The C++ standard specifically states that a template constructor will not be used as a copy constructor. Hence, a default copy constructor is created for such a class while the programmer may be deluded into thinking that the template will be employed for this purpose. [21, Item 5].

**Supports AUTOSAR17 Rule M14-5-2**

**Supports MISRA C++ Rule 14-5-2**

**1790 public base *symbol* of *symbol* has no non-destructor virtual functions**

**info** A public base class contained no virtual functions except possibly virtual destructors. There is a school of thought that public inheritance should only be used to interject custom behavior at the event of virtual function calls. To quote from Marshall Cline, "Never inherit publicly to reuse code (in the base class); inherit publicly in order to be reused (by code that uses base objects polymorphically)" [21, Item 22].

**1791 returned expression begins on the next line**

**info** A line is found that ends with a `return` keyword and with no other tokens following. Did the programmer forget to append a semi-colon? The problem with this is that the next expression is then consumed as part of the `return` statement. Your return might be doing more than you thought. For example:

```
void f( int n, int m ) {
    if( n < 0 ) return    // do not print when n is negative
    print( n );
    print( m );
}
```

Assuming `print()` returns `void`, this is entirely legal but is probably not what you intended. Instead of printing `n` and `m`, for `n` not negative you print just `m`. For `n` negative you print `n`.

To avoid this problem always follow the `return` keyword with something on the same line. It could be a semi-colon, an expression or, for very large expressions, some portion of an expression.

**1793 invoking non-const member function *symbol* of class *symbol* on a temporary**

**info** A non-static and non-const member function was called and an rvalue (a temporary object) of class *symbol* was used to initialize the implicit object parameter. This is legal (and possibly intentional) but suspicious. Consider the following.

```
struct A { void f(); };
...
A().f();           // Info 1793
...
```

In the above the 'non-static, non-const member function' is `A::f()`. The 'implicit object parameter' for the call to `A::f()` is `A()`, a temporary. Since the `A::f()` is non-const it presumably modifies `A()`. But since `A()` is a temporary, any such change is lost. It would at first blush appear to be a mistake.

The Standard normally disallows binding a non-const reference to an rvalue but as a special case allows it for the binding of the implicit object parameter in member function calls. Some popular libraries take advantage of this rule in a legitimate way. For example, the GNU implementation of `std::vector<bool>::operator[]` returns a temporary object of type `std::_Bit_reference` – a class type with a non-const member `operator=()`. `_Bit_reference` serves a dual purpose. If a value is assigned to it, it modifies the original class through its `operator=()`. If a value is extracted from it, it obtains that value from the original class through its `operator bool()`.

This message will not be issued for member functions declared using an rvalue ref-qualifier such as `void f() &&;`.

Probably the best policy to take with this message is to examine instances of it and if this is a library invocation or a specially designed class, then suppress the message with a `-esym()` option.

**1797 assignment operator template *symbol* declared in class *symbol* with no user-declared copy assignment operator**

**info** An assignment operator template with a single parameter whose type is a template parameter type (or a reference thereto) was declared within a class with no user-declared copy assignment operator. Despite its appearance, the template will not be instantiated to act as the assignment operator for the class, and the compiler will still generate a default copy assignment operator unless precluded by other circumstances. For example,

```
struct Y {
    template<typename T>
    Y& operator=(const T&) { }
};
```

will elicit this message. The corresponding message for copy constructors is [1789](#).

**Supports AUTOSAR17 Rule M14-5-3**

**Supports AUTOSAR19 Rule M14-5-3**

**Supports MISRA C++ Rule 14-5-3**

**1798 block scope declaration of *symbol* is taken to mean a member of *symbol* but does not introduce a name**

**info** A block scope function declaration was found within a function whose innermost enclosing namespace was not the global namespace. This alone cannot introduce a namespace member but the declaration of the nested function will still be taken to refer to a (possibly non-existent) member of the innermost enclosing namespace. This can lead to pernicious linker errors if one expects the declared function to introduce a

namespace member into the innermost enclosing namespace or the global namespace. It is also easy to misidentify the innermost enclosing namespace, for example:

```

1 namespace X {
2     namespace Y {
3         struct Z {
4             void f();
5         };
6     }
7 }
8 using X::Y::Z;
9
10 void Z::f() {
11     void g();
12     g();
13 }
```

The declaration on line 11 can be a source of confusion. The enclosing function is defined using the qualified name `Z::f` in a definition written at global scope, but `g` is neither the global `::g` nor the invalid `'Z::g'` (for `Z` is a `struct`). The call to `g` on line 12 will invoke `X::Y::g()`.

This message will not be issued for a block scope function declaration outside of a namespace. See message [9108](#).

Supports AUTOSAR17 Rule M3-1-2

Supports AUTOSAR19 Rule M3-1-2

Supports MISRA C++ Rule 3-1-2

#### 1901 creating a temporary of type *type*

note

PC-lint Plus judges that a temporary needs to be created. This occurs, typically, when a conversion is required to a user object (i.e. class object). Where temporaries are created, can be an issue of some concern to programmers seeking a better understanding of how their programs are likely to behave. But compilers differ in this regard.

#### 1902 unnecessary semicolon follows function definition

note

It is possible to follow a function body with a useless semi-colon. This is not necessarily 'lint' to be removed but may be a preferred style of programming (as semi-colons are placed at the end of other declarations).

#### 1904 old-style c comment

note

For the real bridge-burner one can hunt down and remove all instances of the `/* ... */` form of comment. [6, Item 4]

Supports AUTOSAR17 Rule A2-8-4

#### 1906 exception specification for function *symbol*

note

A function was declared with an exception specification. Some authors contend exception specifications are not worth using due to a presumably false sense of security associated with the specifications. See for example [11, Rule 75].

Supports AUTOSAR17 Rule A1-1-1

Supports AUTOSAR17 Rule A15-4-1

Supports AUTOSAR19 Rule A1-1-1

Supports AUTOSAR19 Rule A15-4-1

**1907 implicit non-trivial destructor generated for *symbol***

**note** The named class does not itself have an explicit destructor but either had a base class that has a destructor or has a member class that has a destructor (or both). In this case a destructor will be generated by the compiler. [13, Section 12.4]

**1908 destructor *symbol* is implicitly virtual due to virtual destructor of base class *symbol* but is not explicitly marked virtual**

The destructor cited was inherited from a base class with a virtual destructor. This word 'virtual' was omitted from the declaration. It is common practice to omit this keyword when implied. See also [1909](#).

**1909 'virtual' assumed; see function *symbol***

**note** The named function overrides a base class virtual function and so is virtual. It is common practice to omit the `virtual` keyword in these cases although some feel that this leads to sloppy programming. This message allows programmers to detect and make explicit which functions are actually virtual.  
Supports MISRA C++ Rule 10-3-2

**1911 implicit call of converting constructor *symbol***

**note** The *symbol* in the message is the name of a constructor called to make an implicit conversion. This message can be helpful in tracking down hidden sources of inefficiencies. [13, Section 12.1]

**1912 implicit call of conversion function from class *symbol* to type *type***

**note** A conversion function (one of the form *symbol*::operator *type* ()) was implicitly called. This message can be helpful in tracking down hidden sources of inefficiencies.  
Supports AUTOSAR19 Rule A13-5-3

**1914 default constructor *symbol* not referenced**

**note** A default constructor was not referenced. When a member function of a class is not referenced, you will normally receive an Informational message ([1714](#)) to that effect. When the member function is the default constructor, however, we give this Elective Note instead. This message is not issued for library symbols and is suppressed for unit checkout (`-unit_check` option).

The rationale for this different treatment lay in the fact that many authors recommend defining a default constructor as a general principle. Therefore, if you are following a modus operandi of not always defining a default constructor you may want to turn on message 1914 instead.

Supports AUTOSAR17 Rule M0-1-10

Supports AUTOSAR19 Rule M0-1-10

Supports MISRA C++ Rule 0-1-10

**1915 virtual function *symbol* overrides function *symbol* and is not marked with 'override'**

**note** A virtual function that overrides a base class function was not declared with the `override` *virt-specifier*. This message is only emitted for C++11 and higher. See also the softer [9421](#) which is only issued if neither `override` nor `final` are specified.

**1916 function *symbol* is variadic**

**note** An ellipsis was encountered while processing the prototype of some function declaration. An ellipsis is a way of breaking the typing system of C or C++.

**1919 *symbol* is not a *copy/move* assignment operator**

**note** A member `operator=` was declared but it is neither a copy assignment operator nor a move assignment operator. Assignment operators with specific signatures are considered special member functions. The declaration of a non-copy non-move assignment operator will not prevent the compiler from generating its own implicit copy and move assignment operators.

If this declaration was intended to allow assignment from an object of a different type, a “converting assignment operator” may be supported more generally using a converting constructor unless the construction and assignment of an object have unique semantics that prevent analogous initialization.

**1920 casting to reference type *type***

**note** The ARM [13, Section 5.4] states that reference casts are often ‘misguided’. However, too many programs are openly using reference casts to place such casts in the Informational category.

**1924 use of c-style cast (*type* to *type*)**

**note** A C-style cast was used in C++ code. This can usually be replaced by one of the newer C++ casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, or a combination thereof. [1, Item 2]. This message is not issued for casts to void used to discard values. See [1954](#).

**Supports AUTOSAR17 Rule A5-2-2**

**Supports AUTOSAR19 Rule A5-2-2**

**Supports MISRA C++ Rule 5-2-4**

**1925 *symbol* is public data member**

**note** The indicated *symbol* is a public data member of a class. If the class is introduced with the keyword `struct` the message is not issued. In some quarters the use of public data members is deprecated. The rationale is that if function calls replace data references in the public interface, the implementation can change without affecting the interface. [6, Item 20]

**1926 default constructor implicitly called to initialize field *symbol***

**note** A member of a class (identified by *symbol*) did not appear in the constructor initialization list. Since it had a default constructor this constructor was implicitly called. Is this what the user intended? Some authorities suggest that all members should appear in the constructor initialization list. [6, Item 12]].

**1927 data member *symbol* absent from initializer list for constructor**

**note** A member of a class (identified by *symbol*) did not appear in a constructor initialization list. If the item remains uninitialized through the whole of the constructor, a Warning [1401](#) is issued. Some authorities suggest that all members should appear in the constructor initialization list. [6, Item 12].

**1928 base class *name* absent from initializer list for constructor**

**note** A base class (identified by *symbol*) did not appear in a constructor initialization list. If a constructor does not appear, the default constructor is called. This may or may not be valid behavior. If a base class is missing from the initializer list of a copy constructor (as opposed to some ordinary constructor), a more severe Warning ([1538](#)) is issued. [6, Item 12].

**Supports AUTOSAR17 Rule A12-1-1**

**Supports AUTOSAR19 Rule A12-1-1**

**Supports MISRA C++ Rule 12-1-2**

**1929 non-member function *symbol* returns reference type *type***

**note** A non-member function was found to be returning a reference. This is not normally considered good practice because responsibility for deleting the object is not easily assigned. No warning is issued if the base class has no constructor. [6, Item 23].

**1930 conversion operator *symbol* found**

**note** A non-explicit conversion operator is a member function of the form:

```
operator Type ();
```

This will be called implicitly by the compiler whenever an object (of the class type) is to be converted to type `Type`. Some programmers consider such implicit calls to be potentially harmful leading to programming situations that are difficult to diagnose. See for example [1, Item 5]. Explicit conversion operators declared with the `explicit` keyword are not reported.

**Supports AUTOSAR19 Rule A13-5-2**

**1931 constructor *symbol* can be used for implicit conversions**

**note** A constructor was found that could be used for implicit conversions. For example:

```
class X {
public:
    X(int);
    ...
};
```

Here any `int` (or type convertible to `int`) could be automatically converted to `X`. This can sometimes cause confusing behavior[1, Item 5]. If this is not what was intended, use the keyword '`explicit`' as in:

```
explicit X(int);
```

This will also serve to suppress this message. See also message [9169](#).

**1932 base class *type* is not abstract**

**note** An abstract class is a class with at least one pure virtual specifier. At least one author has argued [1, Item 33] that all base classes should be abstract although this suggestion flies in the face of existing practice.

**1933 call to unqualified virtual function *symbol* from non-static member function**

**note** A classical C++ gotcha is the calling of a virtual function from within a constructor or a destructor. When we discover a direct call from a constructor or destructor to a virtual function we issue Warning [1506](#). But what about indirect calls. Suppose a constructor calls a function that in turn, perhaps through several levels of call, calls a virtual function. This could be difficult to detect. Dan Saks [19] has suggested a compromise Guideline that "imposes few, if any, practical restrictions". The Guideline, implemented by this Elective Note, issues a message whenever an unqualified virtual function is called by any other (non-static) member function (for the same '`this`' object). For example:

```
class X { virtual void f(); void g(); };

void X::g() {
    f();           // Note 1933
    X::f();        // ok -- non virtual call.
}
```

Even if total abstinence is unwarranted, turning on message [1933](#) occasionally can be helpful in detecting situations when constructors or destructors call virtual functions.

**1934** **shift operator *symbol* should be a non-member function**

**note** It has been suggested [6, Item 19] that you should never make a shift operator a member function unless you're defining `ostream` or `istream` (the message is suppressed in these two cases). The reason is that there is a temptation on the part of the novice to, for example, define output to `ostream` as a class member function left shift that takes `ostream` as an argument. This is exactly backwards. The shift operator normally employs the destination (or source) on the left.

On the other hand, if the class you are defining is the source or destination then defining the shift operators is entirely appropriate.

**1937** **static variable *symbol* of type *type* has a non-trivial destructor**

**note** A static scalar whose name is *symbol* has a destructor. Destructors of static objects are invoked in a predictable order only for objects within the same module (the reverse order of construction). For objects in different modules this order is indeterminate. Hence, if the correct operation of a destructor depends on the existence of an object in some other module an indeterminacy could result. See also [1544](#).

**1938** **constructor *symbol* accesses global data**

**note** A constructor is accessing global data. It is generally not a good idea for constructors to access global data because order of initialization dependencies can be created. If the global data is itself initialized in another module and if the constructor is accessed during initialization, a 'race' condition is established. [6, Item 47]  
**Supports MISRA C++ Rule 12-8-1**

**1939** **casting from base class *type* to derived class *type***

**note** A down cast is a cast from a pointer (or reference) to a base class to a pointer (or reference) to a derived class. A cast down the class hierarchy is fraught with danger. Are you sure that the alleged base class pointer really points to an object in the derived class? Some amount of down casting is necessary, but a wise programmer will reduce this to a minimum. [6, Item 39]  
**Supports AUTOSAR17 Rule M5-2-2**  
**Supports AUTOSAR19 Rule M5-2-2**  
**Supports MISRA C++ Rule 5-2-2**

**1940** **address of non-const reference parameter *symbol* transferred outside of function**

**note** The address of a reference parameter is being transferred (either via a `return` statement, assigned to a static, or assigned through a pointer parameter) to a point where it can persist beyond the lifetime of the function. These are all violations of the Linton Convention (see Murray [20]).

The particular instance at hand is with a reference to a non-const and, as such, it is not considered as dangerous as with a reference to a `const`. (See [1780](#), [1781](#) and [1782](#) for those cases). For example:

```
int *f( int &n ) { return &n; }
int g();
int *p = f( g() );
```

would create a problem were it not for the fact that this is diagnosed as a non-lvalue being assigned to a reference to non-const.

**Supports MISRA C++ Rule 7-5-3**

**1941 *string* assignment operator *symbol* does not return *type***

**note** The typical use of an assignment operator for class `C` is to assign new information to variables of class `C`. If this were the entire story there would be no need for the assignment operator to return anything. However, it is conventional to support chains of assignment as in:

```
C x, y, z;
...
x = y = z;
// parsed as x = (y = z);
```

For this reason assignment normally returns a reference to the object assigned the value. For example, assignment `(y = z)` would return a reference to `y`.

Since it is almost never the case that this variable is to be reassigned, i.e. we almost never wish to write:

```
(x = y) = z; // unusual
```

as a general rule it is better to make the assignment operator return a `const` reference. This will generate a warning when the unusual case is attempted.

But experts differ. Some maintain that in order to support non-const member functions operating directly on the result of an assignment as in:

```
(x = y).mangle();
```

where, as its name suggests, `mangle` is non-const it would be necessary for the return value of assignment to be non-const. Another reason to not insist on the `const` qualifier is that the default assignment operator returns simply a reference to object and not a reference to `const` object. In an age of generic programming, compatibility may be more important than the additional protection that the `const` would offer.

**1943 declaration of *symbol* of type *type* may require global runtime construction**

**note** This message is issued for file-scope variables of class type that have a non-trivial constructor that requires the constructor to be executed to initialize the object at startup time. This can be a potential performance concern.

**1944 declaration of *symbol* of type *type* requires a global destructor**

**note** This message is issued for file-scope variables of class type that have a non-trivial destructor that requires the destructor to be executed to destroy the object at shutdown time. This can be a potential performance concern.

**1945 declaration of *symbol* of type *type* requires an exit-time destructor**

**note** This message is issued for file-scope variables of class type that have a non-trivial destructor that requires the destructor to be executed to destroy the object at shutdown time. This can be a potential performance concern.

**1946 use of functional-style cast to convert from type *type* to type *type***

**note** This message is issued for all functional-style casts except those that apply a converting constructor / conversion operator that has been marked as `explicit`.

Supports MISRA C++ Rule 5-2-4



**1954 use of c-style cast to void (*type to type*)**

**note** A C-style cast to `void` was used in C++ code. If one endeavors to eliminate all C-style casts, a `static_cast` could be used instead.

**Supports AUTOSAR17 Rule A5-2-2**

**Supports AUTOSAR19 Rule A5-2-2**

**1962 member function *symbol* contains deep modification**

**info** The designated member function could be declared `const` but shouldn't be because it contains a deep modification. For example:

```
class X {
    char *p;

public:
    void f() { *p = 0; }
};
```

will elicit this message indicating that `X::f()` contains a deep modification. A modification is considered shallow if it modifies (or exposes for modification) a class member directly. A modification is considered deep if it modifies information indirectly through a class member pointer. This Elective Note is available for completeness so that a programmer can find all functions that could result in a class being modified. It does not indicate that the programming is deficient. In particular, if the function is marked `const` an Info 1763 will be issued. See also [1762](#), [1763](#).

**1966 catch(...) encountered after catch clause**

**note** An ellipsis was used in a catch handler resulting in a handler that will catch any exception. This "catch-all" handler was preceded by one or more catch handlers in the same try block such that this handler will catch any exceptions not caught by one of the more specific handlers. Catch-all exception handlers are generally considered a bad practice due to the inability to distinguish between different types of exceptions and the potential to hide serious issues. The use of such an exception handler without any preceding catch clauses is diagnosed by message [1766](#).

**Supports CWE-396 - Declaration of Catch for Generic Exception**

**1970 use of default capture (*string*) in lambda expression**

**note** This Elective Note diagnoses the use of default capture in a lambda expression. *string* is either `=` or `&`. The use of default capture can have unintended consequences, even in apparently innocuous situations and as such it has been suggested that default capture never be used. For an in-depth discussion of the issue, see [22, Item #1].

**1971 use of function try block for non-constructor function *symbol***

**note** The motivation for the creation of the function-try block in C++ is to allow for the handling of exceptions thrown during the processing of constructor initializer lists. Such exceptions cannot be handled inside of the body of the constructor as the body is not yet entered. While function-try blocks are allowed for non-constructor functions, the same functionality can be obtained using the more general try-catch block inside the body of the function.

**1972 empty declaration**

**note** An empty declaration was encountered; this can happen from an extraneous semi-colon:

```
int x;;
```

Note: In PC-lint this was reported as error 19.

Supports **CERT C MSC12-C** - *Detect and remove code that has no effect or is never executed*

Supports **CWE-561** - *Dead Code*

### 1973 deletion of non-parameter pointer to const

**note** The `delete` operator was applied to a non-parameter pointer to `const`. This is legal and not necessarily suspect. See also message 1726 reports on cases where the pointer being deleted is a function parameter, which is more likely to result in unexpected behavior.

## 22.4 Messages 2000-2999

### 2001 request for *string* integer type with at least *integer* bits could not be processed

**error** An appropriately sized integer type could not be found when attempting to determine the smallest integer type with enough bits to represent a bitfield or an integer constant expression.

### 2005 *hex/octal* escape sequence out of range

**error** The size of a character constant specified with `\xddd` or `\xhhh` equaled or exceeded `2**b` where `b` is the number of bits in a byte (established by the `-sb` option). The default is `-sb8`.

Note that in PC-lint Plus 2.1 and earlier this message was previously emitted as message 541.

### 2006 no hexadecimal digits following *string* escape sequence

**error** A `\x` or `\u` escape sequence was seen but there were no hexadecimal digits immediately following the sequence.  
Supports **MISRA C 2004 Rule 4.1**

### 2400 unexpected internal condition '*string*'

**warning** PC-lint Plus has encountered an unexpected situation while processing the provided source code. This doesn't necessarily represent either a bug in the source code or in PC-lint Plus, and PC-lint Plus will continue to operate normally, but rather serves to report potentially interesting circumstances that may be of use to Vector Informatik GmbH engineering staff. This message is not emitted unless appropriate debugging options are enabled.

### 2401 cannot mix positional and non-positional arguments in format string

**warning** The format string for a `printf/scanf` style function contains both positional and non-positional arguments. Positional arguments are an extension provided by POSIX implementations but mixing positional and non-positional arguments results in undefined behavior. For example:

```
printf("%1$d %d", 1, 2);
```

will elicit this message.

Supports **CERT C FIO47-C** - *Use valid format strings*

Supports **CWE-134** - *Use of Externally-Controlled Format String*

Supports **CWE-685** - *Function Call With Incorrect Number of Arguments*

Supports **CWE-686** - *Function Call With Incorrect Argument Type*

### 2402 '*string*' specified field *string* is missing a matching '*int*' argument

**warning** The format string for a `printf/scanf` style function contains a conversion specifier whose width or precision

is given as an asterisk (\*) indicating that the width/precision be extracted from the next argument, which should have type `int`, but this argument was not provided.

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

#### **2403** **field *string* should have type *type*, but argument has type *type***

**warning**

The width or precision of a conversion specifier within the format for a `printf` or `scanf` style function was specified with an asterisk (\*) and as such a corresponding `int` argument was expected to represent the width/precision but the argument in that position was not the correct type. For example:

```
extern double f;
printf("%*d", f, f);
```

will yield the messages:

```
field width should have type 'int', but argument has type 'double'
```

**Supports CERT C INT00-C** - *Understand the data model used by your implementation(s)*

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

#### **2404** **invalid position specified for *string***

**warning**

Within the format string of a `printf` or `scanf` style function, a positional parameter specifier was expected for a field width or precision that used the asterisk (\*) to indicate that the field or width should be taken from the argument list but one was not provided. For example:

```
printf("%1$d", 1, 2);
```

will yield the message:

```
invalid position specified for field width
```

This is because when positional specifiers are used within a format string, all arguments must have corresponding positional specifiers. The correct way to indicate that the field width corresponds to the second data argument is:

```
printf("%1$*2$d", 1, 2);
```

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

#### **2405** ***string* used with '*string*' conversion specifier is undefined**

**warning**

The use of a field width or precision with an incompatible conversion specifier has been encountered. Standard C allows a precision to be used only with the `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers and a field width to be used with any conversion specifiers except for `n`. Use of field width/precision outside of these conversion specifiers results in undefined behavior.

**Supports CERT C FIO47-C** - *Use valid format strings*

**Supports CWE-134** - *Use of Externally-Controlled Format String*

**Supports CWE-685** - *Function Call With Incorrect Number of Arguments*

**Supports CWE-686** - *Function Call With Incorrect Argument Type*

- 2406** **no closing ']' for '%[' in scanf format string**  
**warning** Within a format string for a `scanf` style function, a `'%['` was seen denoting the start of a scan list but there was no terminating `']'`. The lack of a closing bracket makes the conversion specification invalid and results in undefined behavior.  
**Supports CERT C FIO47-C - Use valid format strings**  
**Supports CWE-134 - Use of Externally-Controlled Format String**  
**Supports CWE-685 - Function Call With Incorrect Number of Arguments**  
**Supports CWE-686 - Function Call With Incorrect Argument Type**
- 2407** **zero field width in scanf format string is unused**  
**warning** Within a `scanf` style function, a zero was given as the maximum field width. Standard C specifies that the maximum field width for `scanf` must be a "decimal integer greater than zero". Providing a zero as the width makes the conversion specifier invalid resulting in undefined behavior.  
**Supports CERT C FIO47-C - Use valid format strings**  
**Supports CWE-134 - Use of Externally-Controlled Format String**  
**Supports CWE-685 - Function Call With Incorrect Number of Arguments**  
**Supports CWE-686 - Function Call With Incorrect Argument Type**
- 2408** **cannot pass *string* object of type *type* to variadic *string*; expected type from format string was *type***  
**warning** A non-POD or non-trivial class type that cannot be passed as a variadic function argument was given as the argument to a `printf/scanf` style function. The first *type* specifies the type of the argument that was provided, the second *type* specifies the type that was expected from the format string.  
**Supports CERT C DCL11-C - Understand the type issues associated with variadic functions**
- 2410** **re-entrant initializer for static local variable *symbol* causes undefined behavior**  
**warning** Recursively executing the initializer for a static local variable is undefined behavior, even if it appears not to cause an infinite loop. An implementation with proper support for thread-safe static initialization is likely to deadlock.  
**Supports CWE-833 - Deadlock**
- 2414** **non-standard literal suffix '*string*'**  
**warning** This message is issued when a non-standard, non-user-defined, numeric literal suffix that is recognized by PC-lint Plus is encountered. The use of non-standard literal suffixes is not portable. The suffixes reported by this message include `q/Q`, `I8/I16/I32/I64` (and the lowercase equivalents), `NaN` and `Infinity`.
- 2415** **loop comparison value '*integer*' out of range for operator '*string*'**  
**warning** A comparison was seen in a loop condition between a loop invariant integer tracked by [Value Tracking](#) and another expression that was determined to be out of range as described in [Precision, Viable Bit Patterns, and Representable Values](#). This message is not issued when the comparison involves a constant expression (see [587](#), [650](#), and [685](#)) or for expressions that are not `for` or `while` conditions (see [696](#)).
- 2416** **generic selection association type *type* will never be eligible for selection**  
**warning** A generic selection was encountered with an association whose type will never be eligible for selection. When types are checked against the association list, top level qualifiers on objects are removed and both arrays and function values are decayed into pointers. Therefore, providing an association for them is not useful and is almost certainly an error. Associations that are not selectable include: a `const`-qualified object type, a

volatile-qualified object type, a restrict-qualified object type, an atomic object type, an array type, a function type, and a structure or union definition.

Supports MISRA C 2012 AMD3 Rule 23.4

**2417** *'string' qualifier on function type type*

**warning** A function type was specified with `const` or `volatile`. Specifying a function with type qualifiers in C will result in undefined behavior. In C++ type qualifiers are ignored. For example:

```
typedef int ftype (void);
const    ftype cfunc; // Warning 2417
volatile ftype vfunc; // Warning 2417
```

Supports MISRA C 2012 AMD3 Rule 17.13

**2418** *declaration of symbol **symbol** with strong type '**strong-type**' conflicts with previous declaration with strong type '**strong-type**'*

A strong type conflict was detected when comparing the strong types (or lack thereof) in a redeclaration to those in the original declaration earlier in the module. Note that such a conflict is not subject to softeners.

**2419** *controlling expression contains a side effect which will not be evaluated*

**warning** The controlling expression of a *generic-selection* has a side effect. The controlling expression is never evaluated and is only used for its type. Thus, side effects are not applied, even if the expression has a type that would cause it to be evaluated by `sizeof` such as a VLA. The related message [9213](#) will report controlling expressions containing function calls. For example:

```
int x = 0;
int foo() { return 0; }
_Generic(x++, int: 1, default: 0); // Warning 2419
_Generic(foo(), int: 1, default: 0); // Note 9213
```

This message is not emitted when the controlling expression is expanded from a macro argument.

Supports MISRA C 2012 AMD3 Rule 23.2

**2423** *apparent domain error for function **symbol**, argument **integer** (value=**string**) outside of accepted range (**string**)*

A value was provided to a mathematical function that will result in a domain error. For example, the `acos` function is only defined for values in the range  $[-1, 1]$ , values provided outside this range will be diagnosed by this message. Value tracking is used to determine the value provided to the function. For example:

```
double foo(double x, double y) {
    acos(x + y);
}
void bar() {
    foo(0.5, 0.75);
}
```

will elicit the message:

```
warning 2423: apparent domain error for function 'acos(double)',
             argument 1 (value=1.25) outside of accepted range (between -1 and 1)
acos(x + y);
      ^
```

Supports AUTOSAR19 Rule A0-4-4

Supports CERT C FLP32-C - Prevent or detect domain and range errors in math functions

**Supports CWE-391** - *Unchecked Error Condition*

**Supports CWE-682** - *Incorrect Calculation*

**Supports CWE-687** - *Function Call With Incorrectly Specified Argument Value*

**2425** user-defined function semantic '*string*' was rejected during call to function *symbol* because *string*  
warning

A call was made to a function for which a user-defined semantic exists but the semantic could not be applied because it contains a semantic that is not valid for this call. There are several reasons this can occur including specifying a semantic for an argument that does not exist, a return value of a type that conflicts with the actual return value, or the use of a symbol or macro in the semantic that cannot be resolved at the time of the call.

**2426** return value (*string*) of call to function *symbol* conflicts with return semantic '*string*'  
warning

A user-defined return semantic was specified for a function for which PC-lint Plus has access to the implementation. Furthermore, PC-lint has determined that during a specific call of the function the actual value returned conflicts with the claimed return value in the return semantic. This represents a likely error in either the implementation of the function or the specification of the semantic. See also [9.2.2.2 Return Semantic Validation](#) in the Semantics chapter.

**2427** *initializer\_list* elements will be destroyed before returning  
warning

The array associated with an initializer list is allocated with a temporary lifetime. The lifetime of the array will not be extended beyond the full expression of a return statement. The returned initializer list will contain dangling pointers. For example:

```
#include <initializer_list>

std::initializer_list<int> f() {
    return { 1, 2, 3 }; // The memory used to store the array
                       // elements will be freed before returning.
}

void g() {
    auto x = f();
    // Attempting to access the elements of x will read invalid memory.
}
```

Note that this message will be issued when returning a local variable of *initializer\_list* type (regardless of initialization source), including a parameter. While it may initially appear safe to return an *initializer\_list* argument as the lifetime was determined by the caller, this is likely to lead to invalid use of the return value in the caller when a temporary argument is destroyed at the end of the statement of the call expression.

**2430** missing whitespace between macro name *name* and definition  
warning

Standard C requires the presence of whitespace between a macro name and its definition for object-like macros. For example:

```
#define MINUS-
```

will elicit:

```
warning 2430: missing whitespace between macro name 'MINUS' and definition
#define MINUS-
~
```

Despite the warning, the macro MINUS is still defined to - so it is safe to suppress this message for legacy code that cannot be changed. The best way to address the warning is to place a space between the macro name and definition:

```
#define MINUS -
```

**2431** *#line/GNU line directive starting with zero is not interpreted as an octal number*  
**warning** The line number provided to the `#line` *number* preprocessing directive (and the GNU equivalent `# number`) is always interpreted as a decimal number, even when the first digit is a zero. For example, `#line 034` is treated as `#line 34`, not as `#line 28` (the decimal equivalent of octal 34). As such, a `#line` directive with a line number beginning with a zero is suspicious.

**2432** *macro `name` used as header guard is followed by a #define of a similar but different macro*  
**warning** *'name'*  
 Within a construct that appears to be a macro include guard, the name of the macro being checked is similar to, but different from, the name of the macro subsequently defined. For example:

```
#ifndef FOO_INCLUDED
#define FOO_INCLOSED
...
#endif
```

will elicit:

```
warning 2432: macro 'FOO_INCLUDED' used as header guard is followed by a
      #define of a similar but different macro 'FOO_INCLOSED'
#ifndef FOO_INCLUDED
      ^~~~~~
```

This usually represents a typo, which will prevent the include guard from functioning as intended. This message can be suppressed with `-estring` using the name of the macro being defined, e.g. `-estring(2432, FOO_INCLOSED)` if the difference was intentional.

**2433** *conversion specifier '`string`' is not allowed for bounds-checked format function*  
**warning** Bounds-checked format functions are described in Annex K of the C11 standard. The bounds-checked `printf`-like functions forbid the use of the `%n` conversion specifier.

**2434** *memory was potentially deallocated*  
**warning** This message is a less certain variant of [449](#) and is issued when the deallocation was dependent on conditional execution flow at runtime.  
**Supports CERT C MEM00-C** - *Allocate and free memory in the same module, at the same level of abstraction*  
**Supports CERT C MEM30-C** - *Do not access freed memory*  
**Supports CWE-415** - *Double Free*  
**Supports CWE-416** - *Use After Free*  
**Supports CWE-666** - *Operation on Resource in Wrong Phase of Lifetime*  
**Supports CWE-672** - *Operation on a Resource after Expiration or Release*  
**Supports CWE-758** - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*  
**Supports CWE-825** - *Expired Pointer Dereference*

**2435** *duplicate '`string`' declaration specifier*  
**warning** The same declaration specifier was used more than once in the declaration of a symbol. For example:

```
inline inline void foo();
```

will elicit this message. Was this intended? While legal, it is suspect. Other specifiers that will be diagnosed for duplicates include `virtual`, `explicit`, `_Noreturn`, `friend`, and `constexpr`.

**2436** *function symbol declared 'noreturn' should not return*

**warning**

A function that was declared as not returning either with the keyword `_Noreturn` or a GCC or C++11 `noreturn` attribute contained a return statement. Returning from a function designated as not returning invokes undefined behavior.

**Supports AUTOSAR19 Rule A7-6-1**

**Supports MISRA C 2012 AMD3 Rule 17.9**

**2437** *indirection of non-volatile null pointer may be optimized out*

**warning**

An indirection on a non-volatile null pointer was encountered. While this is undefined behavior as far as Standard C is concerned, the programmer may have intended for this to generate a trap condition relying on implementation details but the compiler is likely to simply remove the offending indirection instead. The null pointer should be volatile to indicate to the compiler that it should not be optimized out.

**2438** *comparing values of different enumeration types (*type* and *type*)*

**warning**

The values of two different enumeration types were used in an equality or comparison operation. This is suspect because there is no intrinsic relationship among different enumeration types and as such it usually doesn't make sense to compare them. For example:

```
enum color { RED, GREEN, BLUE };
enum fruit { APPLE, PEAR, MANGO };

void foo(enum color c, enum fruit f) {
    if (c == f) return;    // 2438 issued here
    // ...
}
```

The message is parameterized by the two enumeration types compared.

**2439** *lint comment does not contain any options*

**warning**

A lint comment was encountered that did not contain any lint options, was this a mistake? The comment may be empty or may start with text that does not begin an option. For example:

```
//lint e714 -e715
```

Since `e714` does not start with a `-`, `+`, or `!`, it, along with everything that follows, is assumed to be commentary. In this case `-e714` was probably meant.

**2440** *string 'string' in comparison is never null*

**warning**

The address of a function, array, or variable was directly compared to null. This is suspicious because the address of a function or variable can never be null in well-formed code. Note that this message is not given for null checks of function or object pointers. For example:

```
void foo(int *pi) {
    if (!pi) return;        // Okay
    if (&pi == 0) return;    // 2440
    if (foo != 0) return;    // 2440
}
```



The first *string* parameter is one of 'function', 'array', or 'address of' and the second *string* parameter represents the corresponding function, array, or variable.

**Supports CERT C EXP16-C** - *Do not compare function pointers to constant values*

**2441** *string 'string' used in boolean context is never null*

**warning** The address of a function, array, or variable was used in a boolean context. This is suspicious because such an address can never be false. Note that this message is not given for function or object pointers. For example:

```
void foo(int *pi) {
    if (!pi) return;    // Okay
    if (&pi) return;    // 2441
    if (!foo) return;   // 2441
}
```

The first *string* parameter is one of 'function', 'array', or 'address of' and the second *string* parameter represents the corresponding function, array, or variable.

**Supports CERT C EXP16-C** - *Do not compare function pointers to constant values*

**2444** *case value is not in enumeration type*

**warning** The condition of a switch statement has `enum` type but contains a `case` statement with a value that doesn't correspond to any of the enumerators in the `enum`. For example:

```
enum color { RED, GREEN, BLUE };
void foo(enum color c) {
    switch (c) {
        case RED:        // OK
        case RED + 1:     // OK, refers to GREEN
        case 2:           // OK, refers to BLUE
        case 3: ...       // Warning 2444, no member with value 3
    }
}
```

**2445** *cast from type to type increases required alignment from integer to integer*

**warning** A cast was made from a pointer to one type to a pointer to a type that has greater alignment requirements than the type pointed to by the original pointer. For example, assuming an alignment requirement of 4 bytes for 'int' and 8 bytes for 'long double':

```
void foo(int *pi) {
    long double *pld = (long double *)pi;
}
```

will result in the message:

```
warning 2445: cast from 'int *' to 'long double *' increases
    required alignment from 4 to 8
long double *pld = (long double *)pi;
                  ^~~~~~
```

Accessing the value through the new pointer may invoke undefined behavior if it is not properly aligned. The alignment requirements of fundamental types can be set using the `-a` option.

The message is parameterized by the types of the pointer before and after the cast and the alignment requirements of the types before and after the cast.

**Supports CERT C EXP36-C** - *Do not cast pointers into more strictly aligned pointer types*

**2446 pasting formed '*string*', an invalid preprocessing token**

**warning** During a token pasting operation performed by the preprocessor `##` operator, an invalid token was formed. This is illegal even if the result is immediately pasted with another token that would then form a valid token. For example, a naive token concatenation macro might look like:

```
CAT(x, y) x##y
```

which would work fine in cases like `int i = CAT(1,2);` and expand to `int i = 12;` without issue. The problem comes about when the macro is invoked recursively, such as:

```
int i = CAT(CAT(1, 2), 3);
```

which will be greeted with:

```
warning 2446: pasting formed '3', an invalid preprocessing token
int i = CAT(CAT(1, 2),3);
      ^
supplemental 893: expanded from macro 'CAT'
#define CAT(x,y) x##y
      ^
```

followed by other parsing errors. One way to handle this is to use two macros:

```
#define CATX(x, y) x##y
#define CAT(x, y) XCAT(x, y)
```

**2447 'main' function should not be declared as '*string*'**

**warning** This message is issued when the `main` function is declared as `static`, `inline`, `constexpr`, or `deleted`. The C++ Standard forbids the `main` function to be declared with these specifiers.

**2448 'main' function should return type 'int'**

**warning** According to the C Standard, the `main` function must return `int` in a hosted environment but a return type other than `int` was specified for `main`. If you are targeting a freestanding/embedded platform or making use of non-standard extensions, you should suppress this message.

**2449 *string* discards qualifiers**

**warning**

**2450 null character ignored**

**warning** A literal null character was encountered within the module being processed and will be ignored by PC-lint Plus.

**2452 *string* converts between pointers to integer types *string***

**warning** A pointer to an signed integer type was implicitly converted to or from a pointer to the corresponding unsigned integer type. For example:

```
void foo(int *p) {
    unsigned *up = p;    // Warning 2452
}
```

**Supports CWE-135 - Incorrect Calculation of Multi-Byte String Length**

**2453 incompatible pointer to integer conversion *string string***

**warning** A pointer type was implicitly converted to an incompatible integer type. For example:

```
void foo(float *p) {
    int i = p;  // Warning 2453
}
```

**2454 incompatible pointer types *string string***

**warning** A pointer type was implicitly converted to an incompatible pointer type. For example:

```
void foo(float *pf) {
    int *pi = pf;  // Warning 2454
}
```

**Supports CERT C STR38-C - Do not confuse narrow and wide character strings and functions**

**Supports MISRA C 2012 Rule 1.3**

**Supports MISRA C 2004 Rule 1.2**

**Supports CWE-135 - Incorrect Calculation of Multi-Byte String Length**

**2455 incompatible function pointer types *string string***

**warning** A function pointer type was implicitly converted to an incompatible function pointer type. For example:

```
void foo(int i) {
    int (*pf)(float) = &foo;  // Warning 2455
}
```

**2456 C++ language linkage specification encountered in C mode**

**warning** A C++ language linkage specification was encountered in a C module. For example:

```
extern "C++" int i;
```

This may indicate that a C++ module is incorrectly being processed in C mode or that a region of code that is only intended to be processed in C++ is not properly guarded (e.g. with `#ifdef __cplusplus`). Language linkage specifications in C mode are supported by some embedded compilers. If your compiler supports this, feel free to suppress this message.

**2457 non-recursive mutex '*name*' locked recursively in function '*name*' of thread '*name*' at *location***

**warning** While processing the call graph during [Thread Analysis](#), PC-lint Plus detected multiple lock attempts of a non-recursive mutex without an intervening unlock operation. The lock attempts may occur within a single function or may span multiple functions (possibly in different modules) or involve recursive calls into a function. Attempting to lock a non-recursive mutex multiple times will result in a deadlock in most mutex implementations. Supplemental messages provide the call path from the thread root function to the point where the mutex was locked recursively.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports CWE-833 - Deadlock**

- 2460** *string* literal *string* provided as argument *integer* to function *symbol*  
**warning** A string, character, integer, or floating literal was provided as an argument to a function parameter which was designated with the [noliteral](#) argument semantic indicating that the argument should not be a literal. For example, perhaps a database connection function should not receive a string literal in the password argument field.  
**Supports CERT C MSC32-C** - Properly seed pseudorandom number generators  
**Supports CERT C MSC41-C** - Never hard code sensitive information  
**Supports CWE-798** - Use of Hard-coded Credentials
- 2461** *'rand/random'* function used without any explicit call to *'srand/srandom'*  
**warning** This message is issued at the conclusion of global wrap-up if a call to the standard `rand` or POSIX `random` function was found in the program but no call to the corresponding seed function `srand` or `srandom` was found.  
**Supports CERT C MSC32-C** - Properly seed pseudorandom number generators  
**Supports CWE-327** - Use of a Broken or Risky Cryptographic Algorithm  
**Supports CWE-330** - Use of Insufficiently Random Values  
**Supports CWE-331** - Insufficient Entropy  
**Supports CWE-337** - Predictable Seed in Pseudo-Random Number Generator (PRNG)
- 2462** mutexes *'name'* and *'name'* have lock order mismatch in function *'name'* of thread *'name'* at *location* and function *'name'* of thread *'name'* at *location*  
**warning** The specified mutexes are locked in different orders in different call paths (possibly spanning different modules). The call path from each thread root to the conflicting lock orders are provided as supplemental messages as are the final lock orders within each thread. Inconsistent order of mutex acquisition can result in deadlocks between threads.  
 This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).  
**Supports CERT C CON35-C** - Avoid deadlock by locking in a predefined order  
**Supports CWE-833** - Deadlock
- 2463** *'string'* statement while locked  
**warning** A `goto` or label statement (indicated by *string*) was encountered while a mutex was locked. The use of `goto` is best avoided while holding a mutex as doing so presents an entire class of potential issues. Processing will continue ignoring the `goto` or label statement for the purpose of thread analysis. If a `goto` jumps to a point with a different mutex lock state (e.g. jumps over a mutex lock or unlock operation), the corresponding thread analysis may be incorrect. This message may be safely suppressed in cases where jumps do not traverse such operations. This message is accompanied by one or more supplemental messages providing the locations at which the currently held mutexes were locked.
- 2465** redefinition of tag *type* will not be visible outside of this function  
**warning** A tag that was previously defined is being redefined in a function parameter list. While this is legal, it is suspect as this redefinition will only be visible within the function. It would be better to use another name and/or place the desired definition outside the function if the intention is to make the tag visible elsewhere.
- 2466** *symbol* was used despite being marked as *'unused'*  
**warning** The specified symbol was used despite being marked as unused, either via `#pragma unused`, the GCC `__attribute__` syntax, or with a C++11-style attribute specified. For example:

```
int i = 1;
#pragma unused(i)
int j [[gnu::unused]] = 2;
int k __attribute__((unused)) = 3;
```

Message 2466 will be issued if `i`, `j`, or `k` are subsequently used.

#### 2467 multiple definitions of function *symbol*

**warning** Multiple definitions of the specified function were found which would impair thread analysis as PC-lint Plus does not know which definition to use. A link error would presumably occur during compilation of the program as well.

Attempted issuance of this message will disable inter-module thread analysis (see [Inhibition of Thread Analysis](#)).

#### 2468 semantic mismatch for function *symbol*

**warning** The specified function was referenced in multiple modules for which either 1) different thread names were attributed to the function via the `thread` or `thread_mono` semantic or 2) different thread-protected statuses were specified for the function via the `thread_protected` semantic.

Attempted issuance of this message will disable inter-module thread analysis (see [Inhibition of Thread Analysis](#)).

#### 2470 multiple consecutive pushbacks onto stream is not portable

**warning** A file stream was subjected to multiple consecutive pushbacks (such as by the `ungetc` function). Standard C guarantees support for pushing back a single character. While some implementations support pushback of multiple characters, this is non-portable.

**Supports CERT C FIO13-C** - *Never push back anything other than one read character*

#### 2471 operation on stream that has been closed

**warning** An operation was performed on a file stream that has been closed. The value of a file pointer after calling `fclose` is indeterminate and attempts to perform file operations in such a pointer result in undefined behavior.

**Supports CERT C FIO46-C** - *Do not access a closed file*

**Supports MISRA C 2012 Rule 22.6**

**Supports CWE-666** - *Operation on Resource in Wrong Phase of Lifetime*

**Supports CWE-672** - *Operation on a Resource after Expiration or Release*

**Supports CWE-758** - *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*

**Supports CWE-910** - *Use of Expired File Descriptor*

**Supports CWE-1341** - *Multiple Releases of Same Resource or Handle*

#### 2472 non-standard file mode *character/component* '*string*' in mode string '*string*'

**warning** A non-standard character was encountered in the *mode* argument to `fopen` or `freopen`. The allowable characters are `a`, `b`, `r`, `w`, `x`, and `+`; other characters appearing in the *mode* argument results in undefined behavior.

**Supports CERT C FIO11-C** - *Take care when specifying the mode parameter of fopen()*

#### 2473 file mode string '*string*' is not a legal file open mode: *string*

**warning** The *mode* argument in a call to `fopen` or `freopen` contained valid characters but was not one of the

specific combinations allowed by Standard C and therefore will result in undefined behavior. The allowed combinations are:

|       |       |        |       |
|-------|-------|--------|-------|
| "r"   | "w"   | "wx"   | "a"   |
| "rb"  | "wb"  | "wbx"  | "ab"  |
| "r+"  | "w+"  | "w+x"  | "a+"  |
| "r+b" | "w+b" | "w+bx" | "a+b" |
| "rb+" | "wb+" | "wb+x" | "ab+" |

The second *string* parameter provides an explanation of why the *mode* argument is invalid.

**Supports CERT C FIO11-C** - *Take care when specifying the mode parameter of fopen()*

**2474 attempt to flush stream that isn't open for writing**

**warning** An file stream that was not opened for writing was provided as an argument to a file-flushing function such as `fflush`. Attempting to flush a file stream that wasn't opened for writing (or update) results in undefined behavior.

**2475 attempt to flush stream after an input operation**

**warning** An attempt was made to flush a file stream immediately following an input operation; this results in undefined behavior.

**2476 attempt to perform read operation on stream not opened for reading**

**warning** An attempt was made to read from a file stream that was not opened for reading. This results in undefined behavior.

**Supports CWE-910** - *Use of Expired File Descriptor*

**2477 attempt to perform write operation on stream not opened for writing**

**warning** An attempt was made to write to a file stream that was not opened for writing. This results in undefined behavior.

**Supports MISRA C 2012 Rule 22.4**

**Supports CWE-910** - *Use of Expired File Descriptor*

**2478 attempt to perform read operation on stream after write without an intervening flush or reposition**

An attempt was made to read from a file stream following a write operation on the stream. Failure to flush or reposition the stream between a write and subsequent read operation results in undefined behavior.

**Supports CERT C FIO39-C** - *Do not alternately input and output from a stream without an intervening flush or positioning call*

**Supports CWE-664** - *Improper Control of a Resource Through its Lifetime*

**2479 attempt to perform write operation on stream after read without an intervening reposition**

**warning** An attempt was made to write to a file stream following a read operation on the stream. Failure to reposition the stream between a read and subsequent write operation results in undefined behavior.

**Supports CERT C FIO39-C** - *Do not alternately input and output from a stream without an intervening flush or positioning call*

**Supports CWE-664** - *Improper Control of a Resource Through its Lifetime*

**2480 attempt to perform byte-oriented operation on stream following wide-stream operation**  
**warning** A byte-oriented file manipulation function was called with a wide-oriented file stream which results in undefined behavior.  
**Supports CERT C STR38-C** - *Do not confuse narrow and wide character strings and functions*

**2481 attempt to perform wide-oriented operation on stream following byte-stream operation**  
**warning** A wide-oriented file manipulation function was called with a byte-oriented file stream which results in undefined behavior.  
**Supports CERT C STR38-C** - *Do not confuse narrow and wide character strings and functions*

**2482 enumeration constant '*name*' not defined**  
**warning** This message is issued when a call to `mtx_init` is seen without a definition for `mtx_recursive` or a call to `mtx_trylock` is seen without a definition for `thrd_success`. Without knowing the value of `mtx_recursive`, PC-lint Plus will not be able to determine whether the mutex being initialized by `mtx_init` is recursive. Without a definition for `thrd_success`, PC-lint Plus will not be able to infer the lock status of the mutex after a call to `mtx_trylock`.

**2483 invalid semantics for '*name*', *string***  
**warning** This message is issued when invalid function thread semantics are specified using the `-sem` option. This can occur if a semantic is employed that requires additional semantics to be specified which were missing or if multiple conflicting semantic options are specified.

The *name* parameter contains the name of the function specified in the offending semantic and the *string* parameter contains additional information about the invalid nature of the semantic.

For example:

```
-sem(foo, thread, thread_unsafe)
```

will elicit:

```
warning 2483:  invalid semantics for 'foo', thread_unsafe incompatible with thread
```

since it does not make sense for a function to have both the `thread` semantic and the `thread_unsafe` semantic.

None of the semantics specified in the reported `-sem` option will be applied.

**2484 invalid semantics for argument *string* of '*name*', *string***  
**warning** An invalid mutex/thread argument semantic was provided. For example:

```
-sem(f, locker_lock, mutex_locker(1), mutex_locker(2))
```

will elicit:

```
warning 2484:  invalid semantics for argument 2 of 'f', too many mutex_locker
```

because only one argument of a function can be designated as a `mutex_locker`. None of the semantics specified in the reported `-sem` option will be applied. This message is similar to [2483](#) but reports on function *argument* semantics.

Note that the combination of semantics that contribute the the invalid collective need not appear in the same option. For example:

```
-sem(f, locker_lock, mutex_locker(1))
-sem(f, mutex_locker(2))
```

will elicit message 2484 for the second `-sem` option. In this case, only the semantic specified in the second `-sem` option will be ignored (not applied).

**2485** **invalid semantics for locker class *symbol*, *string***  
**warning** A class was designated as a *locker class* using the `-sem` option. A property of the class (specified in the *string* parameter) makes it ineligible to be a locker class. The value of the *string* parameter is one of:

- has copy constructor
- has copy assignment operator
- has move constructor without a default constructor
- has move assignment operator without a default constructor
- has locker\_release without a default constructor
- has mutex\_tag\_defer and/or mutex\_tag\_try\_to\_lock without a default constructor
- no valid constructor
- no destructor
- has bad move constructor

This message is issued the first time a call to any member function of the class is encountered. Supplemental messages will communicate the member functions whose locker semantics will be removed.

**2486** **invalid semantics for *symbol*, *string***  
**warning** Invalid thread function argument semantics were detected for the specified function. The possible values of the *string* parameter and their corresponding meanings are:

- missing `mutex_attribute`  
 A `mutex_attribute_destroy`, `mutex_attribute_initialize`, or `mutex_attribute_set` semantic was specified for the function without a corresponding requisite `mutex_attribute` semantic.
- missing `mutex_is_recursive`, `mutex_is_shared`, or `mutex_attribute`  
 The `mutex_initialize` semantic was specified for the function without also specifying at least one of the `mutex_is_recursive`, `mutex_is_shared`, or `mutex_attribute` semantics.
- missing `mutex_locker`  
 A [locker class semantic](#) requiring one or two `mutex_locker` argument semantics was specified without the requisite number of `mutex_locker` semantics.
- missing `mutex` or `mutex_remaining`  
 A `mutex_destroy`, `mutex_initialize`, `mutex_initialize_std_c`, `mutex_lock`, `mutex_lock_shared`, `mutex_unlock`, `mutex_unlock_shared`, `mutex_validate`, or `locker_create` semantic was specified for the function without a corresponding requisite `mutex` or `mutex_remaining` semantic.
- must be void  
 A `thread_lock` semantic was specified for a function which was not declared as returning void.
- cannot be `try_lock_none`  
 One of the semantics `mutex_lock`, `mutex_lock_shared`, `locker_lock`, or `locker_lock_shared` was specified for the function which is declared as returning a non-void value. Such a function is considered



to be a *trylock*-like function and a [trylock semantic](#) other than `try_lock_none` (the default) must be specified as well.

This message is issued for the first call to the reported function within a module. Mutex/thread semantics will be removed from this function as indicated by the accompanying supplemental message.

**2488** *mutex **symbol** passed more than once*  
**warning** The same mutex was passed multiple times (directly or indirectly via a locker object) to a locking function or locker class constructor. This will likely result in undefined behavior and/or a deadlock for a non-recursive mutex and is suspicious at best for a recursive mutex.  
**Supports** **CWE-833** - *Deadlock*

**2489** *exclusively locked mutex **symbol** is being shared **locked/unlocked***  
**warning** A shared locking or unlocking operation is being attempted on a mutex which is already exclusively locked by the function. Attempting to share lock or unlock a `std::shared_mutex` is undefined if the same thread holds an exclusive lock on the mutex. In other threading libraries this is likely to result in a deadlock.  
**Supports** **CWE-833** - *Deadlock*

**2490** *symbolic constant '**name**' not defined*  
**warning** A call to the `pthread_mutexattr_settype` function was seen at a point where neither of the symbolic constants `PTHREAD_MUTEX_RECURSIVE` nor `PTHREAD_MUTEX_RECURSIVE_NP` were defined with a value that PC-lint Plus could extract. At least one of these symbolic constants should be defined as either an enumeration constant or an object-like macro. If defined as an enumeration constant, make sure the definition is visible by PC-lint Plus (e.g. by including the appropriate header). If defined as a macro, the macro definition should be an integer literal. If `PTHREAD_MUTEX_RECURSIVE` or `PTHREAD_MUTEX_RECURSIVE_NP` are defined as something other than an integer literal, the `++d` option can be used to override the definition to the appropriate integer literal, e.g. `++dPTHREAD_MUTEX_RECURSIVE=1`. The `-mutex_attr` option can also be used to inform PC-lint Plus of values and bitmasks used with `pthread_mutexattr_settype` that indicate a recursive mutex attribute.

**2491** *unknown expression '**string**' in sizeof will evaluate to 0, use -pp\_sizeof to change the value used for evaluation*  
**warning** A `sizeof` expression was encountered inside of a preprocessor conditional. Furthermore, the expression appearing within `sizeof` was not previously registered with the `-pp_sizeof` option and will evaluate to zero. See the `-pp_sizeof` option for more information.

**2492** *locker **symbol** already locked*  
**warning** An attempt is being made to lock an already-locked locker class instance. For example:

```
#include <mutex>

std::mutex m;

void foo() {
    std::unique_lock locker(m);
    locker.lock();      // 2492 - locker already locked
}
```

A supplemental message will provide the location of the previous lock.

**2493** **locker *symbol* is not locked**

**warning** An attempt was made to unlock a locker class instance that is not currently locked.  
**Supports** **CWE-832** - *Unlock of a Resource that is not Locked*

**2494** **mutex *symbol* is not locked**

**warning** A locker class was constructed in a way that requires a locked mutex but a mutex that was not locked by the current function was provided instead. For example:

```
#include <mutex>

std::mutex m;

void foo() {
    std::lock_guard guard(m, std::adopt_lock);
}
```

The `std::lock_guard` constructor taking `std::adopt_lock` causes the lock guard to take ownership of an already acquired lock but `m` was not locked prior to constructing `guard` which results in undefined behavior.

**2495** **mutex *symbol* is not *locked/unlocked***

**warning** A mutex was passed as an argument to a function with either the `mutex_must_be_locked` or `mutex_must_be_unlocked` semantic but PC-lint Plus was able to determine that the mutex being passed was not in the correct lock state expected by the function.

**2496** **locker *symbol* is not *locked/unlocked***

**warning** A locker class object was passed as an argument to a function with either the `mutex_must_be_locked` or `mutex_must_be_unlocked` semantic but PC-lint Plus was able to determine that the locker being passed was not in the correct lock state expected by the function.

**2498** **comparison of object representations of floating point values of member *symbol* of type *type* may produce unexpected results that differ from a value equality test**

**warning** Bitwise comparison of structures containing floating point values leads to the comparison of the object representations of data members of floating point type. See [2499](#).

**Supports** **AUTOSAR17 Rule M3-9-3**

**Supports** **AUTOSAR19 Rule M3-9-3**

**Supports** **CERT C FLP37-C** - *Do not use object representations to compare floating-point values*

**Supports** **MISRA C++ Rule 3-9-3**

**2499** **comparison of object representations of floating point values of type *type* may produce unexpected results that differ from a value equality test**

**warning** Bitwise comparison of the object representations of floating point values may differ in a number of ways from value comparison using comparison operators. Common differences include inequality of representations of zero and negative zero and equality of identical NaN representations (but inequality of NaNs with different payload bits).

**Supports** **AUTOSAR17 Rule M3-9-3**

**Supports** **AUTOSAR19 Rule M3-9-3**

**Supports** **CERT C FLP37-C** - *Do not use object representations to compare floating-point values*

**Supports** **MISRA C++ Rule 3-9-3**

**2501** **warning** negation of value of unsigned type *type* yields a value of signed type *type* due to integral promotion

An unsigned integer type was promoted to a signed type as an operand to the unary minus operator. This may surprise those who are otherwise familiar with the common adage that applying unary minus to an unsigned type does not yield a negative value (see message 501). For example: (assuming 16-bit shorts and 32-bit ints)

```
-(unsigned)5; // 2^32 - 5, type is still unsigned int
-(unsigned short)5; // -5, type is signed int
```

**2502** **warning** differing alignment requirements seen for *symbol* (*string* vs *string*)

An externally visible object was declared with different explicit alignment requirements. If an object with external linkage has a conflicting alignment requirement, the behavior is undefined.

Supports MISRA C 2012 AMD3 Rule 8.15

**2504** **warning** argument of type *type* supplied to type-generic macro '*macro*' resulting in call to function *symbol* should have a *string* type

The operand argument passed to a type-generic macro has an inappropriate type. The operand argument must have a signed, unsigned, or floating (real or complex) type. Type-generic macros from `tgmath.h` can support real types, complex types, or both. Passing a complex type to a macro expecting a real type results in undefined behavior. Likewise, arguments of non-arithmetic types are not convertible to any of the corresponding real types defined for the macros and attempting to use them is also undefined behavior.

```
float f;
_Complex float cf;
atan2(f); // OK
atan2(&f); // Warning 2504
atan2(cf); // Warning 2504
```

**2505** **warning** argument to integer constant macro '*macro*' *string*

The argument passed to the specified integer constant macro had an inappropriate form. The argument must be an unsuffixed integer literal whose value does not exceed the exact-width type of the macro used. Constant expressions are also not allowed. The behavior of such arguments to integer constant macros is undefined. The *string* parameter describes how the supplied argument is inappropriate.

```
INT32_C(-100); // Warning 2505
-INT32_C(100); // OK
```

Supports MISRA C 2012 AMD3 Rule 7.5

**2511** **warning** `try_lock` return value was discarded

The return value of a *trylock*-like function was discarded. A *trylock*-like function is one which may fail if the lock cannot be acquired and whose return value indicates whether locking was successful. If the return value of such a function is ignored, there is no way for the program to know (or PC-lint Plus to infer) if the lock was obtained and subsequent operations that make assumptions about the success of the call may be incorrectly performed.

Supports CWE-362 - Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

Supports CWE-667 - Improper Locking

**2512 try\_lock return value was modified**

**warning** The result of a *trylock*-like function was stored but the value was later modified. For example:

```
#include <mutex>

std::mutex m;

void foo(bool b) {
    bool result = m.try_lock();
    result = b;
    if (result) {
        m.unlock();
    }
}
```

Since the **result** variable no longer represents whether the mutex was acquired, PC-lint Plus will not be able to properly associated the actions predicated on the value of **result** with the lock state of the mutex.

**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**2513 try\_lock return value was manipulated**

**warning** The result of a *trylock*-like function was *manipulated*. By *manipulated* we mean any use that is not a direct test of its value in a way that PC-lint Plus can appropriately inference the result such as with the binary **==** or **!=** operators or the **!** unary operator. Assigning the value to a variable is permitted and similar manipulation of such a variable will also be reported. The value that represents the return of a *trylock*-like function should be tested in a direct way in order to determine whether the lock request succeeded. See also [2511](#) and [2512](#).  
**Supports CWE-362** - *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

**2520 mutex attribute *symbol* is not initialized**

**warning** An uninitialized mutex attribute was used in a call to a function with the **mutex\_attribute\_destroy**, **mutex\_initialize**, or **mutex\_attribute\_set** semantic. By default, the relevant functions employing these semantics are the POSIX pthread functions **pthread\_mutexattr\_destroy**, **pthread\_rwlockattr\_destroy**, **pthread\_mutex\_init**, **pthread\_rwlock\_init**, and **pthread\_mutexattr\_settype**. Passing an uninitialized mutex attribute in a call to one of these functions invokes undefined behavior and will impair the ability of PC-lint Plus to determine whether a mutex initialized using the mutex attribute represents a recursive mutex.

**2521 mutex attribute *symbol* is already destroyed**

**warning** A mutex attribute object that has already been destroyed is being passed to a function with the **mutex\_attribute\_destroy** semantic. By default, the functions with this semantic are the POSIX pthreads functions **pthread\_mutexattr\_destroy** and **pthread\_rwlockattr\_destroy**. Attempting to use a destroyed mutex attribute in any way other than by reinitializing it via an initialization function (e.g. **pthread\_mutexattr\_init** or **pthread\_rwlockattr\_init**) invokes undefined behavior.

**2522 mutex attribute *symbol* is already initialized**

**warning** A function with the **mutex\_attribute\_initialize** semantic is being called with a mutex attribute argument that has already been initialized. By default, the functions with this semantic are the POSIX pthread functions **pthread\_mutexattr\_init** and **pthread\_rwlockattr\_init**. Attempting to initialize an already initialized mutex attribute object with one of these functions results in undefined behavior.

**2530 mutex '*name*' has inconsistent types**

**note** A mutex was initialized as both recursive and non-recursive or as both a shared and a non-shared mutex at different locations. The locations where the mutex is initialized are provided in supplemental messages. PC-lint Plus will assume the mutex is recursive if it is ever initialized as recursive and shared if it is ever initialized as shared.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**2531 mutex '*name*' has usage incompatible with its type '*string*'**

**note** The specified mutex was initialized as a non-recursive mutex but was locked recursively or was initialized as a non-shared mutex but was used with a shared lock function. The type of the mutex is provided in the *string* parameter and the locations at which the incompatible operations take place are provided in supplemental messages.

This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).

**Supports MISRA C 2012 AMD4 Rule 22.18**

**2536 illegal character encoding in string literal**

**warning** Source files are expected to be encoded as UTF-8 or UTF-16 text. The provided source file was presumed to contain UTF-8 text but an invalid byte sequence was encountered inside of an unprefix string literal. Encoding errors encountered outside of string literals are reported via error [336](#).

**2540 mutex *symbol* used as a locker**

**warning** The specified *symbol* was previously employed as a mutex but is now being used as a locker. This can occur when incorrect function argument semantics have been applied.

**2586 *string name* is deprecated**

**warning** This message is issued when an entity is encountered that has been deprecated using either the C++14 deprecated attribute or the GCC deprecated attribute syntax. The type and name of the deprecated entity are provided in the message. If the deprecation contains a reason text, this is included as an additional string parameter as the end of the message. An [891](#) message provides the location of the actual deprecation. For example:

```
[[deprecated]] void foo();

void bar() {
    foo();
}
```

The use of `foo` on line 4 results in the message:

```
warning 2586: Function 'foo' is deprecated
    foo();
    ^
supplemental 891: Function 'foo' was marked deprecated here
    [[deprecated]] void foo();
    ^
```

This message is not used to report the use of entities that are deprecated with the `-deprecate` option, such instances are instead reported by message [586](#).

**2601** passing address of auto variable *symbol* to parameter *integer* of function *symbol*

**warning** A pointer to automatic storage was passed as an argument to a function parameter with the `no_ptr_to_auto` semantic. Note that during a specific walk this message is not limited to local variables in the immediate function performing the call, for example:

```

1 //lint -sem(save, no_ptr_to_auto(1))
2 void save(int* p);
3 void g(int* x) {
4     save(x);
5 }
6 void f(void) {
7     int a = 5;
8     g(&a);
9 }
```

will report:

```

4 warning 2601: passing address of auto variable 'a' to parameter 1 of
  function 'save'
  save(x);
  ~
8 supplemental 894: during specific walk g(&(5))
  g(&a);
  ~
```

**Supports CERT C POS34-C** - Do not call `putenv()` with a pointer to an automatic variable as the argument

**Supports CWE-562** - Return of Stack Variable Address

**Supports CWE-686** - Function Call With Incorrect Argument Type

**2618** non-type specifier '*string*' appears after a type

**warning** A non-type specifier was provided after a type specifier in a declaration. For example `int inline foo();` will elicit this message. The non-type specifiers reported by this message are: `friend`, `constexpr`, `thread_local`, `mutable`, `inline`, `virtual`, and `explicit`. See also message [618](#) which reports storage class specifiers appearing after a type and message [963](#) which reports on `const` and `volatile` qualifiers before or after a type.

**Supports AUTOSAR17 Rule A7-1-8**

**Supports AUTOSAR19 Rule A7-1-8**

**2623** possible domain error for function *symbol*, argument *integer* (value=*string*) outside of accepted range (*string*)

**warning** Value tracking inferencing has determined that the value provided to a mathematical function is within a range that contains values that are not appropriate for the function and may result in a domain error. For example:

```

double foo(unsigned i) {
    if (i <= 10)
        acos(i);
}
```

will solicit the message:

```

warning 2623: possible domain error for function 'acos(double)', argument 1
  (value=0:10) outside of accepted range (between -1 and 1)
  acos(i);
  ~
```

as the valid range for the argument to `acos` is `[-1, 1]` and all that is known about the value provided is that it is between 0 and 10. To eliminate the diagnostic, the test should be corrected as in

```
if (i <= 1)
```

**Supports** **CWE-687** - *Function Call With Incorrectly Specified Argument Value*

**2641** **implicit conversion of enum *symbol* to floating point type *type***

**warning** An enumeration type was implicitly converted to a floating point type. Since enumerations are always represented using integral underlying types, it is suspicious to use an enumeration value in a floating point context. This message can be suppressed by using a cast.

**2650** **constant '*integer*' out of range for '*string*' portion of compound comparison operator '*string*'**

**warning** This message is issued when only the "greater than" or "less than" part of a "greater than or equal" or "less than or equal" compound comparison operator is out of range. For example (assuming 8-bit bytes):

```
1 void foo(unsigned char a) {
2     if (a == 255) { }    // Okay - 'a' could be equal to 255
3     if (a > 255) { }     // 650 - 'a' can't be greater than 255
4     if (a >= 255) { }    // 2650 - 'a' could be equal but not greater than 255
```

Message 2650 is issued on line 4 because while `a` could be equal to 255, it cannot be greater than 255 so the use of the `>=` operator is suspicious (perhaps `a` was intended to be compared to a different value). See also message [650](#) which is issued when the provided constant is out of range for the entire comparison operator.

**Supports** **CERT C MSC21-C** - *Use robust loop termination conditions*

**2662** **pointer arithmetic on pointer that may not refer to an array**

**warning** This message is issued instead of [662](#) when a pointer that appears likely not to refer to an array is subject to integer arithmetic. Addition, subtraction, and array subscripting are considered. Referring to the value itself with the operand zero is ignored. For example:

```
void f(int a) {
    int* p = &a;
    p[0] = 0;
    p[1] = 0;    // Warning 2662
    p + 0;
    p + 1;       // Warning 2662
}
```

**Supports** **CERT C ARR37-C** - *Do not add or subtract an integer to a pointer to a non-array object*

**Supports** **CWE-119** - *Improper Restriction of Operations within the Bounds of a Memory Buffer*

**2666** **expression with side effects passed to unexpanded parameter *integer* of macro '*string*': *detail***

**warning** This message is issued when a function-like macro is invoked with a parameter that appears as though it would have side effects *if* it were evaluated but since the corresponding parameter is not expanded in the macro definition, no side-effect will occur. E.g.:

```
#define DEBUG_VAL(x)

int process(int i) {
    DEBUG_VAL(++i); // 2666 - increment doesn't occur
    /* ... */
    return i;
}
```

Since the parameter isn't expanded, `++i` is not evaluated and the increment does not occur which may be unexpected. If the intention is that the side-effect occurs regardless of how the macro is defined, the expression provoking the side effect should be placed outside the macro invocation. For the purpose of this message, any expression appearing to contain a function call is considered to have side-effects.

*Detail* is one of “parameter is not referenced in the expansion” or “parameter is only used with `##` operators”. The message can be suppressed based on the value of the *detail* parameter by using `-estring`.

Supports CERT C PRE31-C - *Avoid side effects in arguments to unsafe macros*

- 2670** **call to `async-signal-unsafe` function *symbol* within *signal-handler-category symbol***  
**warning** A [signal handler](#) called an [async-signal-unsafe](#) function. This is likely to cause undefined behavior.  
 Supports CERT C SIG30-C - *Call only asynchronous-safe functions within signal handlers*  
 Supports CWE-364 - *Signal Handler Race Condition*  
 Supports CWE-479 - *Signal Handler Use of a Non-reentrant Function*
- 2671** **returning from exception signal handler *symbol***  
**warning** An [exception signal handler](#) contained a `return` statement. This is likely to cause undefined behavior.  
 Supports CERT C SIG35-C - *Do not return from a computational exception signal handler*
- 2701** ***variable/function symbol* declared outside of header is not defined in the same source file**  
**info** The specified *symbol* was declared inside of a module but not defined inside the same module. If the *symbol* is defined in another module, it would be better to place the declaration of the *symbol* in a header and include that header in the modules that use the *symbol*.
- 2702** **static symbol *symbol* declared in header not referenced**  
**info** The named static symbol was declared in a header included by the module but was not used within the including module. If the symbol had been declared in the module itself, warning [528](#) would be issued instead.
- 2703** **dangling else, add braces to body of parent statement to make intent explicit**  
**info** A dangling `else` occurs when an `if/else` construct appears as the unbraced body of an `if` statement. In such cases, it may not be clear which of the `if` statements the `else` is intended to be associated with. For example:

```
int foo(int a, int b) {
    if (a)
    if (b)
        return 1;
    else
        return 0;
    return 2;
}
```

Is the `else` statement part of the `if (a)` or the `if (b)`? In C and C++, the `else` is associated with the closest preceding `if` that it is legal to be associated with so the `else` in the example is associated with `if (b)`. The message can be addressed by placing braces around the parent `if (a)` statement to make the intention explicit:

```
int foo(int a, int b) {
    if (a) {
        if (b)
```



```

        return 1;
    else
        return 0;
    }
    return 2;
}

```

**2704 potentially negating the most negative number**

**info** An integer value with the potential to equal the most negative possible integer was negated. In a two's complement representation, there is no positive equivalent to the most negative representable integer. For example:

```

void f(int a) {
    if (a < 0) {
        a = -a;
    }
    // Not safe to assume a is non-negative, negation of -2147483648
    // yields the same negative value in many compilers.
}

```

**Supports CERT C INT08-C** - *Verify that all integer values are in range*

**Supports CERT C INT16-C** - *Do not make assumptions about representation of signed integers*

**2705 type qualifier(s) 'string' applied to return type *has/have* no effect**

**info** A type qualifier was provided for the return type of a function but has no effect. Was the intention to qualify the function, a pointe type, a reference to the type returned, or something else? For example:

```
const int foo();
```

will be met with this message as `const` qualifier has no effect in this context.

**2706 integer constant value does not match any enumerator in enumeration *type***

**info** An integer constant is being used to assign a value to an enumeration type but the constant value does not match the value of any of the enumeration's enumerators. For example:

```

enum color { RED, GREEN, BLUE };

void foo(enum color);
void bar() {
    enum color c1 = RED;    // Okay
    enum color c2 = 0;      // Okay
    enum color c3 = 3;      // 2706
    foo(4);                 // 2706
}

```

The values 3 and 4 are not part of the enumeration 'color' so **2706** will be issued in these cases.

**2707 function *symbol* could be declared as 'noreturn'**

**info** The specified function has no means of returning to its caller but this information is not included in the functions declaration via either the C11 `_Noreturn` keyword, the C++11 `noreturn` attribute, or the GCC `noreturn` attribute. Adding this information to the declaration may help clarify the purpose of the function.

**Supports MISRA C 2012 AMD3 Rule 17.11**

**2708 pointer to variably modified array type *type* used in declaration of *symbol***

**info** A variable having pointer to a variably-modified array type was declared. For example,

```
int n = 10;
int (*a)[n];
```

will produce this message. If two pointers to arrays are used in a manner that requires them to be compatible and the size specifiers are not the same, this will result in undefined behavior. Since variably modified arrays cannot have a known size at compile time, the risk of mismatched size specifiers is increased.

**Supports MISRA C 2012 AMD4 Rule 18.10**

**2709 array subscript is of type 'char'**

**info** A value of type 'char' was used as the subscript to an array. 'char' is a signed type on some platforms, relying on the signedness of 'char' in this way is not wise.

**2712 large pass-by-value parameter *symbol* of type *type* (*integer* bytes) for function *symbol***

**info** The specified function was declared as taking a large object type by-value. It may be more efficient to have the function receive a pointer or reference instead. The threshold for determining what constitutes a large object is specified using the `-size` option.

**2713 large return type *type* (*integer* bytes) for function *symbol***

**info** A large object type is being returned by-value from the specified function; you might want to consider returning the object by pointer or reference instead. The threshold for determining what constitutes a large object is specified using the `-size` option.

**2715 token pasting of ',' and `__VA_ARGS__` is a GNU extension**

**info** The token pasting operator `##` appeared between a comma and the `__VA_ARGS__` macro. While supported by several compilers as a mechanism by which to elide a trailing comma in a variadic macro, such a construct is technically undefined and could result in different behavior on a compiler that doesn't support this extension. See the discussion for the `fr` macro for more details.

**2716 tentative array definition for variable *symbol* assumed to have one element**

**info** This message is issued when a declaration for a variable of array type that acts as a tentative definition is encountered without a declared array size. A *tentative definition* in C is a file-scope declaration without an initializer that does not contain an `extern` storage class specifier. If the translation unit contains no external definition for an identifier, the C Standard specifies that it is defined with the composite type of the tentative definition(s) for that identifier. In the case of an array without a size, this becomes an array with one element. This might represent an oversight in the program. If this was intentional, it would be clearer to define the array explicitly with one element.

**2751 indeterminable *string***

**info** During thread analysis, an entity of the type specified by *string* was encountered in a context for which PC-lint Plus was unable to determine its identity. *String* is one of `locker`, `locker state`, `locker tag name`, `mutex`, `mutex attribute`, or `thread root function`. For example, this message will be issued if the thread root function provided in a call to a thread-creation function is a function pointer whose value cannot be inferred by PC-lint Plus. This message is also issued for uses of local mutexes, PC-lint Plus currently expects mutexes to have static storage duration. This message is issued for with a *string* value of `locker state` if

the state of a locker object is unknown when referenced.

**Supports MISRA C 2012 AMD4 Rule 22.13**

## 2752 incompatible semantics for locker class *symbol*, corrections applied

**info** Incompatible thread semantics were detected for one or more member functions of the specified locker class. Corrections to address the incompatible semantics are automatically applied, the details of which are provided in supplemental messages. The semantics specified for the member functions of the locker class should be reviewed and corrected as appropriate.

## 2753 mutex *symbol* may not be *locked/unlocked*

**info** A mutex was passed as an argument to a function with either the `mutex_must_be_locked` or `mutex_must_be_unlocked` semantic but the mutex being passed may not be in the correct state expected by the function. See also message 2495 which will report cases where the mutex is known to be in the wrong lock state.

## 2760 '*srand/srandom*' seeded with '*time*'

**info** This message is issued to report a call to the standard `srand` or POSIX `srandom` function if the seed argument is the result of a call to the standard `time` function. While this is not uncommon, it is considered insecure. The current time is predictable and the poor granularity measured in seconds on typical implementations makes it easy to probe the entire window in which the application may have called the seed function. The current time can also be manipulated in a variety of ways, not only through local access but also potentially through updates from a malicious time server. The ability for an attacker to predict the sequence of pseudorandom numbers that will be generated can render various algorithms insecure.

**Supports CERT C MSC32-C** - Properly seed pseudorandom number generators

**Supports CWE-327** - Use of a Broken or Risky Cryptographic Algorithm

**Supports CWE-330** - Use of Insufficiently Random Values

**Supports CWE-331** - Insufficient Entropy

**Supports CWE-337** - Predictable Seed in Pseudo-Random Number Generator (PRNG)

## 2761 call to non-async-signal-safe function *symbol* within *signal-handler-category symbol*

**info** A [signal handler](#) called a function that was not explicitly [async-signal-safe](#). This could potentially lead to undefined behavior.

**Supports CERT C SIG30-C** - Call only asynchronous-safe functions within signal handlers

**Supports CWE-364** - Signal Handler Race Condition

**Supports CWE-479** - Signal Handler Use of a Non-reentrant Function

## 2762 call to signal registration function *symbol* within *signal-handler-category symbol*

**info** A [signal handler](#) called a [signal registration](#) function. This could lead to undefined behavior if the signal number being registered is not the same signal that invoked this signal handler.

**Supports CERT C SIG34-C** - Do not call `signal()` from within interruptible signal handlers

**Supports CWE-364** - Signal Handler Race Condition

**Supports CWE-479** - Signal Handler Use of a Non-reentrant Function

## 2763 call to signal registration function *symbol* within *signal-handler-category symbol* to register itself

A [signal handler](#) called a [signal registration](#) function to register itself. This could potentially lead to undefined behavior if the signal number being registered is not the same signal that invoked this signal handler.

**Supports CERT C SIG34-C** - Do not call `signal()` from within interruptible signal handlers

**Supports CWE-364** - *Signal Handler Race Condition*

**Supports CWE-479** - *Signal Handler Use of a Non-reentrant Function*

- 2764** **exception signal handler *symbol* does not explicitly end the program**  
**info** An [exception signal handler](#) did not appear to unconditionally end the program. (The end of the function body was reachable for the purposes of message [527](#).) This could potentially lead to undefined behavior.  
**Supports CERT C SIG35-C** - *Do not return from a computational exception signal handler*
- 2765** **reference to variable *symbol* which is neither atomic nor volatile sig\_atomic\_t within *signal-handler-category symbol***  
**info** A [signal handler](#) referenced a non-local variable that was not `_Atomic`, `std::atomic<T>`, nor volatile `sig_atomic_t`. This is likely to cause undefined behavior.  
**Supports CERT C SIG31-C** - *Do not access shared objects in signal handlers*  
**Supports CWE-364** - *Signal Handler Race Condition*  
**Supports CWE-662** - *Improper Synchronization*
- 2770** **type of mutex '*name*' is incomplete '*string*' but its usage requires specific type**  
**note** A mutex was locked recursively or was used with a shared-locking function but the mutex was not initialized in a way that PC-lint Plus could verify the recursive or shared property of the mutex. PC-lint Plus will assume that the mutex is a valid recursive or shared mutex based on its usage. Supplemental messages will provide the locations where the mutex is initialized and the locations where the mutex was locked or unlocked in shared mode or locked recursively.
- This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).
- 2771** **type of mutex '*name*' is incomplete '*string*'**  
**note** A mutex was used but PC-lint Plus was unable to deduce the shared and/or recursive properties of the mutex. This will occur if the initialization(s) do not provide the information necessary to determine if the mutex is shared or recursive. Supplemental messages will provide the locations where the mutex is initialized.
- This message is suppressed if inter-module thread analysis is disabled (see [Inhibition of Thread Analysis](#)).
- 2865** ***string***  
**info** A `#pragma message` directive was encountered. The text of this message is the string provided in the pragma.
- 2901** **stack usage information: *detail***  
**note** When generating stack reporting data, if the [fun](#) flag is set, this message will be issued once for each function in the stack report with *detail* containing the stack information for the function. See [-stack](#) for details.
- 2902** **discarded instance of post-*string* operator**  
**note** A post-increment or post-decrement operator was applied to a scalar in a context where the value will not be used (i.e. as an expression statement, as the left operand to the comma operator, or as the third clause of a `for` statement). A modern optimizing compiler is virtually guaranteed to elide the wasteful copy implied by such an operation when the value is not used but some may prefer to use pre-increment or pre-decrement operators instead for clarity and consistency. See also [1757](#) for potentially more serious cases involving user-defined types.

**2903 initializer list has a chained designator and at least one other field without a designator**

**note** There was an initializer list in which there exists a chained designator and at least one field that was not initialized using a designator. For example,

```
struct Point2D {
    int x;
    int y;
};

struct Point4D {
    int w;
    struct Point2D s;
    int z;
};

struct Point4D my_point = {
    1,
    .s.x = 2,
    3,
    4,
};
```

This message only takes into consideration one level of nested initializers at a time. It is not a violation when, taken as a whole, inner and outer nesting levels have mixed positional initializers or chained designators. Only when a single level of initializers has a mix.

**Supports MISRA C 2012 AMD4 Rule 9.6**

**2904 chained designator conflicts with usage of positional braced initializer list for the same sub-object**

**note**

A sub-object was initialized

1. via braced initializer list in which at least one init expression was positional.
2. At some point before or after this initializer list, a field of the same sub-object was initialized via chained designator in the enclosing init list.

For example,

```
struct Inner {
    int x;
    int y;
    int z;
};

struct Outer {
    struct Inner i;
};

struct Outer o = {
    .i.z = 3,
    .i = {1,2},
};
```

will produce this message.

**Supports MISRA C 2012 AMD4 Rule 9.6**

**2920 unlocking order mismatch**

**note** A set of mutexes was unlocked in an order inconsistent with the locking order. For example:

```
#include <mutex>

std::mutex m1, m2;
int my_data = 0;

void foo(int i) {
    m1.lock();
    m2.lock();

    ++my_data;

    m1.unlock(); // 2920: unlocked before m2
    m2.unlock();
}
```

While it is generally considered good practice to unlock mutexes in the reverse order in which they were locked, inconsistent unlocking order does not suffer the same potential issues as inconsistent locking order (which is diagnosed by message [2462](#)).

**2932 macro *name* used as header guard is followed by a #define of a different macro '*name*'**

**note** A macro with a name that is different from the header guard macro was defined immediately after the header guard. It is conventional practice to define the macro used in the header guard check but this is not always done for every file; this message can be used to identify those that do not follow this pattern. The more egregious violations (those that define a macro with a very similar name) are flagged by message [2432](#).

**2960 integer constant expression with value *integer* provided as argument *integer* to function *symbol***

**note** An integer constant expression was provided as an argument to a function parameter which was designated with the [noliteral](#) argument semantic indicating that the argument should not be a literal. While an integer constant expression does not necessarily consist of a single literal, it may be suspicious for many of the same reasons as the value is constant. For example, perhaps you do not want `srand` to be seeded with a constant expression to ensure it produces a different sequence on each execution. This message can be enabled or suppressed for specific constant values.

**Supports CWE-327** - *Use of a Broken or Risky Cryptographic Algorithm*

**Supports CWE-330** - *Use of Insufficiently Random Values*

**Supports CWE-331** - *Insufficient Entropy*

**Supports CWE-337** - *Predictable Seed in Pseudo-Random Number Generator (PRNG)*

**22.5 Messages 3000-3999****3401 parameter to move constructor *symbol* is an rvalue reference to `const`**

**warning** There are relatively few valid reasons to declare a move constructor taking an rvalue reference to `const` but this construct could easily be formed by accident due to its similarity to a canonical copy constructor. This message will not be given if the move constructor is deleted, as this is occasionally useful.

**3402 lambda capture default captures 'this' by value**

**warning** The `this` pointer was implicitly captured due to a member access inside a lambda. For clarity and to avoid an accidental use-after-free, some prefer to explicitly specify this in the capture list.

This message is emitted for lambdas where the default capture is by value. Access to a member with default capture by value does not capture the member by value but instead captures the `this` pointer. Use of the member within the lambda is still subject to the lifetime of the object pointed to by the `this` pointer.

This message is considered more severe than message 3702 for default capture by reference because the situation reported by that message is less likely to cause misinterpretation of object lifetime considerations, while the situation reported by message 3402 could easily lead to a use-after-free.

**3403 use of `std::move` on value of forwarding reference type *type*; was `std::forward<string>` intended?**

**warning** A forwarding reference (sometimes referred to as a universal reference) was given as an argument to `std::move`. Either the formation of a forwarding reference instead of an rvalue reference or the use of `std::move` instead of `std::forward` was likely accidental. For example:

```
template<typename T>
void f(T&& t) {
    g(std::move(t)); // might unexpectedly move from the caller's lvalue
}
```

**Supports CWE-843 - Access of Resource Using Incompatible Type ('Type Confusion')**

**3405 *symbol* is specified with C linkage but returns type *type* which is incompatible with C**

**warning** A function was specified as having C language linkage but the function returns a type that is not compatible with C so what is the point in having C linkage?

**3406 incomplete return type *type* for function *symbol* specified with C linkage**

**warning** A function that was specified as having C language linkage has an incomplete return type.

**3407 *symbol* should not return null unless declared with 'throw()' or 'noexcept'**

**warning** The implementation for an operator new function has the possibility to return null but the function is not declared with 'throw()' or 'noexcept'. Operator new functions should never return null except in these cases.

**3408 address of reference in comparison is never null in well-formed C++**

**warning** The address of a reference was directly compared to null. This is suspicious because the address of a reference can never be null in well-formed code

```
void foo(int &i) {
    int &ri = i;
    if (&i == 0) return; // 3408
    if (&i != 0) return; // 3408
}
```

**3409 address of reference in boolean context is never null in well-formed C++**

**warning** The address of a reference was used in a boolean context. This is suspicious because such an address can never be false. For example:

```
void foo(int &i) {
    int &ri = i;
    if (&i) return;    // 3409
    if (&ri) return;    // 3409
    if (!&i) return;    // 3409
}
```

**3410 conversion function converting *type* to itself will never be used**

**warning** A class contains a conversion function that converts to the *type* of the class itself. This is suspicious because such a conversion function will never be called. For example:

```
class X {
    operator X();
};
```

will elicit this message.

**3411 conversion function converting *type* to its base class *type* will never be used**

**warning** A class contains a conversion function that converts to the *type* of its base class. This is suspicious because such a conversion function will never be called. For example:

```
class X { };
class Y : public X {
    operator X();
};
```

will elicit this message.

**3412 *type* has virtual functions but non-virtual destructor**

**warning** A class contains at least one virtual function but has a non-private, not-virtual destructor. Classes that may be used as base classes should always have virtual destructors to ensure that instances of derived classes that are deleted through a pointer to the base class are properly destructed. The declaration of a virtual function implies that this class is meant to be used as a base class and as such it should provide a virtual destructor.

**3413 *delete/destructor* called on non-final *type* that has virtual functions but non-virtual destructor**

**warning** This message is similar to 3412 but while the former warns about the potential problem that could arise from having a non-virtual destructor, this message warns when delete is applied to such an object.

**3414 *delete/destructor* called on *type* that is abstract but has non-virtual destructor**

**warning** This message is similar to 3413 but is reported for abstract types.

**3415 pointer initialized to temporary array**

**warning** A pointer is being initialized with a temporary array, which will be destroyed at the end of the containing expression making it impossible to safely dereference the pointer before assigning a new value to it. For example:

```
struct S { int array[10]; };

void f() {
    int *pi = S().array;    // warning 3415, array pointed to by pi
}
```



```

        // will cease to exist after initialization.
    }

```

**3416 'this' pointer used in boolean context is never null**

**warning** The `this` pointer was used in a boolean context such as:

```
if (this) ...
```

Was this a mistake? The `this` pointer is never null in well-formed C++ so such a test is suspect.

**3417 'this' pointer used in comparison is never null**

**warning** The `this` pointer is explicitly tested for null such as

```
if (this == 0)
```

Was this a mistake? The `this` pointer is never null in well-formed C++ so such a test is suspect.

**3418 'reinterpret\_cast' *string* class *symbol* *string* its *string* *symbol* behaves differently than 'static\_cast'**

**warning** A `reinterpret_cast` was used to cast between a class type and a base class type in an unsafe way; `static_cast` should probably be used instead.

**3419 in-class initializer for static data member *symbol* of type *type* is a GNU extension**

**warning** An in-class initializer for a static data member of floating point type was encountered. This is a non-portable extension.

**3420 extraneous template parameter list in template specialization**

**warning** An extraneous template parameter list was provided in the declaration of a template specialization. For example:

```

template <typename T>
T foo(T);

template<>          // warning 3420
template<>
int foo(int);

```

**3421 *string* template partial specialization contains *string* that cannot be deduced so the specialization will never be used**

**warning** In the partial specialization of a class or variable template, the presence of one or more template parameters that cannot be deduced means that the specialization will never be used. Was this a mistake?

**3423 *case value/enumerator value/non-type template argument/array size/constexpr if cannot be narrowed from type to type***

**warning** A case value, enumerator value, non-type template argument, or array size was provided that cannot be narrowed to the required type. For example:

```

void foo(unsigned u) {
    switch(u) {

```

```

        case -1:           // warning 3423, cannot narrow -1 to unsigned
            break;
        ...
    }
}

template <unsigned char I>
struct S { unsigned char value = I; };
S<300> s;           // warning 3423, cannot narrow 300 to unsigned char

```

**3424 warning** *constexpr/constexpr function/constructor never produces a constant expression*

A function marked as `constexpr` or `constexpr` must contain at least one code path that produces a constant expression to be used in a context where a constant expression is required. The specified function was marked as `constexpr` or `constexpr` but does not ever produce a constant expression so the use of `constexpr/constexpr` is suspect.

**3425 warning** *type type cannot be narrowed to type in initializer list*

Inside an initializer list, a prohibited implicit narrowing conversion would be required to perform the initialization. The prohibited implicit conversion is one that is never allowed in list initialization. For example:

```
int i = { 3.0 };
```

will elicit this message because conversion from a floating point type to an integral type is required to perform the initialization but is not an allowed implicit conversion within an initializer list. The issue can be corrected by correcting the type used in the initializer, casting the type, or not using list initialization.

**3426 warning** *non-constant-expression cannot be narrowed from type type to type in initializer list*

Inside an initializer list, a prohibited implicit narrowing conversion would be required to perform the initialization. The implicit conversion is of a type that is allowed for constant expressions but not for the provided expression. For example:

```
extern int i;
float f = { i; };
```

will elicit this message while:

```
float f = { 3 };
```

will not. The issue can be corrected by correcting the type used in the initializer, casting the type, or not using list initialization.

**3427 warning** *constant expression evaluates to string which cannot be narrowed to type type*

Inside an initializer list there is an implicit conversion of a constant expression to a type where the value cannot be represented exactly, which is prohibited. For example:

```
unsigned char c = { 1234 };
```

will elicit this message, assuming 8-bit `chars`. The message can be avoided by correcting the value, using a cast, or by not employing list initialization.

**3428 warning** *out-of-line declaration of a member must be a definition*

A `class` member that is declared out-of-line (outside of the `class` declaration) must have a definition. For example:

```
class A {
    void foo();
}

void A::foo();
```

The second declaration of `A::foo` must contain a definition.

### 3429 parenthesized initialization of a member array is a GNU extension

**warning** In C++11, an array member can be initialized in a member initialization list using extended initializer syntax, e.g.:

```
class A {
    A() : array { 0 } { };
    int array[10];
}
```

A deprecated GNU extension allowed an array member to be initialized using the syntax for initializing class types:

```
A() : array({ 0 });
```

This parenthesis around the initializer here are not Standard.

### 3430 taking the address of a temporary object of type *type*

**warning** An attempt was made to take the address of a temporary object. For example:

```
struct X { ... };

void foo() {
    &X();          // warning 3430
}
```

### 3431 in-class initializer for static data member of type *type* requires 'constexpr' specifier

**warning** A `const static` data member of integral type may be initialized in its in-class declaration with a constant expression. For non-integral types, the member must be declared with `constexpr`, e.g.:

```
struct A {
    const static int i = 3;
    constexpr static float f = 3.0;
};
```

This message is issued for non-integral static data members with an in-class initializer of a type for which `constexpr` is expected but not provided.

### 3432 invalid suffix on literal, C++ 11 requires a space between literal and identifier

**warning** A literal appeared adjacent to an identifier but there was no space separating the two and the identifier was not a valid suffix for the literal.

### 3450 subtracting value of member *symbol* from the address referred to by the 'this' pointer; use of `->` to access the member may have been intended

**warning** The integral or pointer value of a member of `this`, accessed implicitly through the current object, was

subtracted from the `this` pointer. This is almost certainly a mistake where the `>` in `->` was forgotten. For example:

```

1  struct X {
2      bool value;
3      bool getValue() const {
4          return this-value; // intended to be this->value
5      }
6  };

```

If for some reason explicitly applying a negative offset to the `this` pointer based on a member value is actually desired then the member name can be enclosed in parentheses to avoid confusion.

**3701** use of *symbol* implicitly invokes converting constructor *symbol*; *symbol* could be used [without braces]

info

A `push`, `push_back`, or `insert` function is called in a situation where `emplace_back` or `emplace` could be used instead.

**3702** lambda capture default captures 'this' by value

info

The `this` pointer was implicitly captured due to a member access inside a lambda. For clarity, some prefer to explicitly specify `this` in the capture list.

This message is emitted for lambdas where the default capture is by reference. Access to a member with default capture by reference does not directly capture the member by reference but instead captures the `this` pointer.

This message is considered less severe than message [3402](#) for default capture by value because the situation reported by that message could easily lead to a use-after-free, while the situation reported by message 3702 is less likely to cause misinterpretation of object lifetime considerations.

**3703** ellipsis at this point creates a C-style varargs function

info

An ellipsis was encountered, which was probably intended to declare a function parameter pack but instead declares a variable argument function. For example:

```

template <typename... T>
void foo() {
    bar([] {
        void g(T t...); // warning 3703, probably meant g(T... t);
    }...);
}

```

**3704** empty parentheses here declare a function, not a variable

info

A set of empty parentheses were added to what would otherwise be interpreted as a variable declaration but instead results in the declaration of a function. For example:

```

struct S {
    S(int a = 0) : _a(a) { }
    int _a;
};

void foo() {
    S s1(1); // OK, declares and initializes variable s1
}

```

```
    S s2();      // warning 3704, declares a function s2
}
```

`s2` is interpreted as a function that returns type `S` and takes not arguments, not a zero-initialized variable as presumably intended.

There are multiple ways to force interpretation of a variable, e.g.:

```
S s3(0);      // OK
S s4 = S();   // OK, initialize via temporary
S s5{};       // OK, C++11 uniform initialization
```

See also message [3705](#).

### 3705 parenthetic disambiguation results in function declaration

info

A syntactic construct was encountered that could be interpreted as either a variable declaration or a function declaration (sometimes referred to as the "most vexing parse"). The C++ disambiguation rules require that it be interpreted as a function declaration, which may not be what the programmer intended. For example:

```
struct X { };
struct Y {
    Y(const X&);
};

void foo() {
    Y y(X());   // warning 3705
}
```

Here `y` is interpreted as a function that returns an object of type `Y` and takes a single parameter that is a pointer to a function taking no arguments and returning type `X`. In particular, it is *not* interpreted as a declaration of an object of type `Y` initialized with a temporary of type `X` as was almost certainly intended.

There are several ways to force interpretation of a variable declaration. In C++11 and later the simplest way is to employ uniform initialization syntax, for example any of the following would work:

```
Y y1(X{});    // OK, variable declaration
Y y2{X{}};    // OK, variable declaration
Y y3{X{}};    // OK, variable declaration
```

Prior to C++11, an extra pair of parenthesis can be used to force the desired interpretation, e.g.:

```
Y y4((X{}));  // OK, variable declaration
```

The same issue can appear with casts, for example:

```
void foo(double d) {
    int i( int(d) );    // warning 3705
}
```

In this case, `i` is not a variable initialized with the truncated value of `d` but rather a function returning `int` and taking `int`. In addition to the methods mentioned above to force a variable declaration, the functional cast can be converted to a C-style cast or a named cast, e.g.:

```
int i1( (int) d );      // OK, variable declaration
int i2( static_cast<int>(d) ); // OK, variable declaration
```

### 3706 abstract class *symbol* marked '*final/sealed*'

info

An abstract class (one that contains at least one pure virtual specifier) was marked as `final` or `sealed`

preventing the class from being used as a base class. Since abstract classes cannot be instantiated, what would be the purpose of having abstract class that cannot be inherited from?

### 3707 unknown linkage language '*string*'

**info** C++ defines the language linkage specifiers "C" and "C++". Other specifiers may be supported by compilers as an extension with implementation-defined semantics. This message is issued when a language linkage specifier other than "C" or "C++" is encountered. For example:

```
extern "C" int a;      // Okay
extern "C++" int b;    // Okay
extern "ADA" int c;    // Info 3707
```

### 3901 reference to *data member/member function symbol of symbol* does not use an explicit '*this->*'

**note**

A non-static data member or function was referenced inside of the containing class with an implicit `this` object instead of using `this->member` to access the member. For example:

```
struct A {
    int value;
    int getValue() { return this->value; } // OK, explicit this->
    void setValue(int v) { value = v; }    // note 3901
};
```

Some authors suggest always using `this->` to access members to prevent the potential for confusion when local objects or functions with the same name as a member exist in the same scope. See also message 578, which will be issued if a local symbol is declared that hides a member.

### 3902 thrown object of type *type* is not a class derived from `std::exception`

**note**

This message is issued where a throw-expression initializes an exception object that is not derived from `std::exception`. The point is to have a type that can be caught by

```
catch(std::exception & p)
```

instead of

```
catch(...)
```

This way, in a situation where it's necessary to catch everything, information about the kind of error can at least be logged or translated.

Supports AUTOSAR17 Rule A15-1-1

Supports AUTOSAR19 Rule A15-1-1

### 3903 lambda implicitly returning *[unduced] type [written as 'auto'/'decltype(auto)']* does not explicitly specify return type

**note**

A lambda expression did not explicitly specify a return type. A return type of `auto` or `decltype(auto)` is not considered to be explicitly specified. This message is not emitted for a lambda expression returning `void`.

Supports AUTOSAR17 Rule A5-1-6

Supports AUTOSAR19 Rule A5-1-6

## 22.6 Messages 4000-6999

Messages in the 4xxx, 5xxx, and 6xxx ranges are used by PC-lint Plus to report clang compiler errors. Because these messages are generally self-explanatory syntax errors, detailed descriptions of the individual

messages are not included in this manual.

## 22.7 Messages 8000-8999

### 8000 *string*

**info** Messages in the 8xxx range are reserved for user-defined diagnostics, see [+message](#) for more information.

## 22.8 Messages 9000-9999

### 9001 *octal constant 'string' used*

**note** An octal constant appears in the code. Octal constants may be inadvertently interpreted by engineers as decimal values. This message is not issued for a constant zero written as a single digit.

**Supports AUTOSAR17 Rule M2-13-2**

**Supports AUTOSAR19 Rule M2-13-2**

**Supports CERT C DCL18-C - Do not begin integer constants with 0 when specifying a decimal value**

**Supports MISRA C 2012 Rule 7.1**

**Supports MISRA C++ Rule 2-13-2**

**Supports MISRA C 2004 Rule 7.1**

### 9003 *could define global variable *symbol* within function *string**

**note** A variable was declared at global scope but only utilized within one function. Moving the declaration of this variable to that function reduces the chance the variable will be used incorrectly. This message is suppressed for unit checkout (`-unit_check` option).

**Supports AUTOSAR17 Rule M3-4-1**

**Supports AUTOSAR19 Rule M3-4-1**

**Supports CERT C DCL19-C - Minimize the scope of variables and functions**

**Supports MISRA C 2012 Rule 8.9**

**Supports MISRA C++ Rule 3-4-1**

**Supports MISRA C 2004 Rule 8.7**

### 9004 *object/function *symbol* previously declared*

**note** The named symbol was declared in multiple locations, not counting the point of definition for that symbol. Declaring a symbol in one location and in one file helps to ensure consistency between declaration and definition as well as avoiding the risk of conflicting definitions across modules.

**Supports AUTOSAR17 Rule M3-2-3**

**Supports AUTOSAR19 Rule M3-2-3**

**Supports MISRA C 2012 Rule 8.5**

**Supports MISRA C++ Rule 3-2-3**

**Supports MISRA C 2004 Rule 8.8**

### 9005 *cast drops *detail* qualifier(s)*

**note** A cast attempted to remove the qualifiers from an object to which a pointer points or a reference refers. Doing so can result in undesired or unexpected modification of the object in question and may result in an exception being thrown.

**Supports AUTOSAR17 Rule A5-2-3**

**Supports AUTOSAR19 Rule A5-2-3**

**Supports CERT C EXP05-C - Do not cast away a const qualification**

**Supports MISRA C 2012 Rule 11.8**

Supports MISRA C++ Rule 5-2-5

Supports MISRA C 2004 Rule 11.5

Supports CWE-704 - *Incorrect Type Conversion or Cast*

#### 9006 **'sizeof' used on expression with side effect**

**note** If the operand of the `sizeof` operator is an expression, it is not usually evaluated. Attempting to apply `sizeof` to such an expression can result, therefore, in code one expects to be evaluated actually not being evaluated and the side-effects not taking place. This message is not given if the operand is an lvalue of volatile qualified type and is not a variably-lengthed array.

Supports AUTOSAR17 Rule M5-3-4

Supports AUTOSAR19 Rule M5-3-4

Supports CERT C EXP44-C - *Do not rely on side effects in operands to `sizeof`, `__Alignof`, or `__Generic`*

Supports MISRA C 2012 Rule 13.6

Supports MISRA C++ Rule 5-3-4

Supports MISRA C 2004 Rule 12.3

#### 9007 **side effects on right hand of logical operator, '*string*'**

**note** The right hand side of the `||` and `&&` operators is only evaluated if the left hand side evaluates to a certain value. Consequently, code that expects the right hand side to be evaluated regardless of the left hand side can produce unanticipated results.

Supports AUTOSAR17 Rule M5-14-1

Supports AUTOSAR19 Rule M5-14-1

Supports CERT C EXP02-C - *Be aware of the short-circuit behavior of the logical AND and OR operators*

Supports MISRA C 2012 Rule 13.5

Supports MISRA C++ Rule 5-14-1

Supports MISRA C 2004 Rule 12.4

Supports CWE-768 - *Incorrect Short Circuit Evaluation*

#### 9008 **comma operator used**

**note** The comma operator is thought by some to reduce readability in code.

Supports MISRA C 2004 Rule 12.10 (Req)

Supports AUTOSAR17 Rule M5-18-1

Supports AUTOSAR19 Rule M5-18-1

Supports MISRA C 2012 Rule 12.3

Supports MISRA C++ Rule 5-18-1

Supports MISRA C 2004 Rule 12.10

#### 9009 **possible use of floating point loop counter**

**note** The use of floating point variables as loop counters can produce surprising behavior if the accumulation of rounding errors results in a different number of iterations than anticipated.

Supports CERT C FLP30-C - *Do not use floating-point variables as loop counters*

Supports MISRA C 2012 Rule 14.1

Supports MISRA C 2004 Rule 13.4

#### 9010 **conversion from integer type *type* to pointer type *type***

**note** An expression of integral type (other than a null pointer constant) was converted to a pointer type (other than `void*`).

Supports AUTOSAR17 Rule M5-2-8

Supports AUTOSAR19 Rule M5-2-8



Supports MISRA C++ Rule 5-2-8

Supports CWE-587 - *Assignment of a Fixed Address to a Pointer*

#### 9011 multiple loop exits

**note** More than one `break` statement or `goto` statement is used to terminate a loop. Minimizing the number of exits from a loop is thought by some to reduce visual complexity of the code.

Supports MISRA C 2012 Rule 15.4

Supports MISRA C++ Rule 6-6-4

Supports MISRA C 2004 Rule 14.6

#### 9012 body should be a compound statement

**note** Multiple authors have advised making sure the body of every *iteration-statement* and *selection-statement* be a *compound-statement*. However, no `{` was seen to begin the *compound-statement*.

Supports AUTOSAR17 Rule M6-3-1

Supports AUTOSAR17 Rule M6-4-1

Supports AUTOSAR19 Rule M6-3-1

Supports AUTOSAR19 Rule M6-4-1

Supports CERT C EXP19-C - *Use braces for the body of an if, for, or while statement*

Supports MISRA C 2012 Rule 15.6

Supports MISRA C++ Rule 6-3-1

Supports MISRA C++ Rule 6-4-1

Supports MISRA C 2004 Rule 14.8

Supports MISRA C 2004 Rule 14.9

#### 9013 no 'else' at end of 'if ... else if' chain

**note** An `if...else if` chain was seen without a final `else` statement. Providing such a statement helps to act as an analog to the default case of a `switch-statement`.

Supports AUTOSAR17 Rule M6-4-2

Supports AUTOSAR19 Rule M6-4-2

Supports CERT C MSC01-C - *Strive for logical completeness*

Supports MISRA C 2012 Rule 15.7

Supports MISRA C++ Rule 6-4-2

Supports MISRA C 2004 Rule 14.10

#### 9014 switch without default

**note** A `switch-statement` was found without a `default` case. Providing such a case provides defensive programming.

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.4

Supports MISRA C 2004 Rule 15.3

#### 9015 macro '*name*' appearing in argument *integer* of macro '*name*' is used both with and without '#/##' and is subject to further replacement

**note** In the expansion of a function-like macro, a macro argument was used both as an operand to the stringizing or pasting operators and was also used in a way in which it was subject to further macro replacement. For example:

```
#define M1 123
#define FM(x) ident_ ## x + x
...
```

```
FM(10);      // Okay, 10 is not a macro
FM(M1);      // 9015, M1 both expanded and used with ##
```

The FM macro uses the parameter `x` as an operand to the token pasting operator (where a macro argument would not be expanded) and in a context where a macro argument would be expanded. This example expands to:

```
ident_10 + 10;
ident_M1 + 123;
```

In the second invocation, part of the expansion contains the unexpanded macro and another contains the result of the expanded macro argument. This may be confusing and lead to unexpected results.

**Supports MISRA C 2012 Rule 20.12**

#### 9016 performing pointer arithmetic via *addition/subtraction*

**note**

Array indexing is thought, by some, to be more readily understood and less error prone than other forms of pointer arithmetic.

**Supports AUTOSAR17 Rule M5-0-15**

**Supports AUTOSAR19 Rule M5-0-15**

**Supports MISRA C 2012 Rule 18.4**

**Supports MISRA C++ Rule 5-0-15**

**Supports MISRA C 2004 Rule 17.4**

#### 9017 *incrementing/decrementing* pointer

**note**

While at least one standards organization cautions against using any pointer arithmetic besides array indexing, the use of increment or decrement operators with pointers may represent an intuitive application and illustration of the underlying logic. Consequently, such constructs are separated from message 9016 and placed under this one, allowing a more fine tuning of Lint diagnostics.

**Supports MISRA C 2004 Rule 17.4**

#### 9018 union *symbol* declared

**note**

Depending upon padding, alignment, and endianness of union, as well as the size and bit-order of their members, the use of unions can result in unspecified, undefined, or implementation defined behavior, prompting some to advise against their use.

**Supports AUTOSAR17 Rule M9-5-1**

**Supports AUTOSAR19 Rule A9-5-1**

**Supports MISRA C 2012 Rule 19.2**

**Supports MISRA C++ Rule 9-5-1**

**Supports MISRA C 2004 Rule 18.4**

#### 9019 declaration of *symbol* before `#include`

**note**

The symbol mentioned in *string* was seen in a module with a subsequent `#include` directive. It can be argued that collecting all `#include` directives at the beginning of the module helps improve code readability and helps reduce risk of undefined behavior resulting from any use of the ISO standard library before the relevant `#include` directive.

**Supports AUTOSAR17 Rule M16-0-1**

**Supports AUTOSAR19 Rule M16-0-1**

**Supports MISRA C 2012 Rule 20.1**

**Supports MISRA C++ Rule 16-0-1**

**Supports MISRA C 2004 Rule 19.1**

- 9020 header file name '*string*' contains non-standard character '*string*'**  
**note** This message is issued when one of the characters ', ', or \ or one of the character sequences /\* or // appears in the header name specified by a #include directive. The behavior of these characters in a #include directive is undefined in C and implementation-defined in C++.  
 Supports AUTOSAR17 Rule A16-2-1  
 Supports AUTOSAR19 Rule A16-2-1  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C 2012 Rule 20.2  
 Supports MISRA C++ Rule 16-2-4  
 Supports MISRA C++ Rule 16-2-5  
 Supports MISRA C 2004 Rule 19.2
- 9021 use of '#undef' is discouraged: '*detail*'**  
**note** The use of the #undef directive can lead to confusion about whether or not a particular macro exists at a randomly given point of code.  
 Supports AUTOSAR17 Rule A16-0-1  
 Supports AUTOSAR19 Rule A16-0-1  
 Supports MISRA C 2012 Rule 20.5  
 Supports MISRA C++ Rule 16-0-3  
 Supports MISRA C 2004 Rule 19.6
- 9022 unparenthesized macro parameter '*string*' in definition of macro '*string*' *string***  
**note** Multiple authors have cautioned against the use of unparenthesized macro parameters in cases where the parameter is used as an expression. If care is not taken, unparenthesized macro parameters can result in operator precedence rules producing expressions other than intended.  
 Supports AUTOSAR17 Rule M16-0-6  
 Supports AUTOSAR19 Rule M16-0-6  
 Supports CERT C PRE01-C - Use parentheses within macros around parameter names  
 Supports MISRA C++ Rule 16-0-6  
 Supports MISRA C 2004 Rule 19.10
- 9023 multiple use of stringize/pasting operators in definition of macro *name***  
**note** Multiple use of such operators is thought by some to increase risk of undefined behavior.  
 Supports AUTOSAR17 Rule M16-3-1  
 Supports AUTOSAR19 Rule M16-3-1  
 Supports MISRA C 2012 Rule 1.3  
 Supports MISRA C++ Rule 16-3-1  
 Supports MISRA C 2004 Rule 19.12
- 9024 *pasting/stringize* operator used in definition of *object-like/function-like* macro '*string*'**  
**note** The use of token pasting (##) and stringizing (#) preprocessor operators is thought by some to reduce code clarity and increase the risk of undefined behavior.  
 Supports AUTOSAR17 Rule M16-3-2  
 Supports AUTOSAR19 Rule M16-3-2  
 Supports CERT C PRE05-C - Understand macro replacement when concatenating tokens or performing stringification  
 Supports MISRA C 2012 Rule 20.10  
 Supports MISRA C++ Rule 16-3-2  
 Supports MISRA C 2004 Rule 19.13

**9025 more than two levels of pointer indirection**

**note** Three or more levels of pointer indirection may make it harder to understand the code.

Supports AUTOSAR17 Rule A5-0-3

Supports AUTOSAR19 Rule A5-0-3

Supports MISRA C 2012 Rule 18.5

Supports MISRA C++ Rule 5-0-19

Supports MISRA C 2004 Rule 17.5

**9026 function-like macro, 'macro', defined**

**note** Multiple authors have expressed reasons why a function, when possible, should be used in place of a function-like macro.

Supports AUTOSAR17 Rule A16-0-1

Supports AUTOSAR19 Rule A16-0-1

Supports CERT C PRE00-C - *Prefer inline or static functions to function-like macros*

Supports MISRA C 2012 Dir 4.9

Supports MISRA C++ Rule 16-0-4

Supports MISRA C 2004 Rule 19.7

**9027 *essential-type* value is not an appropriate *string* operand to *operator***

**note** Out of concern for unspecified, undefined, and/or implementation defined behavior, some standards urge restrictions on certain types of operands when used with certain operators.

Supports MISRA C 2012 Rule 10.1

**9028 *essential-type* value is not an appropriate *string* operand to *operator***

**note** MISRA C 2012 has defined the concept of *essentially character type* and placed restrictions on the use of expressions with such a type.

Supports MISRA C 2012 Rule 10.2

**9029 *essential-type* value and *essential-type* value cannot be used together as operands to *operator***

**note** MISRA C 2012 has defined the concept of *essentially type* and placed restrictions on the use of expressions with certain types with respect to binary operators.

Supports MISRA C 2012 Rule 10.4

**9030 cannot cast *essential-type* value to *essential-type* type**

**note** MISRA C 2012 has defined the concept of *essential type* and placed restrictions on the use of casts between certain types.

Supports MISRA C 2012 Rule 10.5

**9031 cannot assign a composite expression of type '*essential-type*' to an object of wider type '*essential-type*'**

**note** MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on assignments of the former.

Supports MISRA C 2012 Rule 10.6

**9032 *left/right* operand to *operator* is a composite expression of type '*essential-type*' which is smaller than the *left/right* operand of type '*essential-type*'**

**note** MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on

operands to binary operators when at least one of the operands meets the definition of the former concept.  
**Supports MISRA C 2012 Rule 10.7**

**9033 cannot cast '*essential-type*' to wider/different essential type '*essential-type*'**

**note** MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on casts of the former. This message, when given, is also followed by text explaining why the cast is considered "impermissible".

**Supports MISRA C 2012 Rule 10.8**

**9034 cannot assign '*essential-type*' to narrow/different essential type '*essential-type*'**

**note** MISRA C 2012 has defined the concept of *essential type* and placed restrictions on assignments in relation to such types.

**Supports MISRA C 2012 Rule 10.3**

**9035 variable length array of type *type* declared**

**note** Variable length arrays can introduce unspecified behavior and runtime-dependent undefined behavior. As of C11 it is not required that implementations support this feature. For these reasons, the use of VLAs is often discouraged.

**Supports CERT C ARR32-C - Ensure size arguments for variable length arrays are in a valid range**

**Supports CERT C MEM05-C - Avoid large stack allocations**

**Supports MISRA C 2012 Rule 18.8**

**Supports CWE-758 - Reliance on Undefined, Unspecified, or Implementation-Defined Behavior**

**9036 essential type of condition of '*do/for/if/while*' statement is '*essential-type*' but should be boolean**

MISRA C 2012 has defined the concept of *essentially Boolean* type and requires the conditional expressions of all if and iteration-statements comply with this definition.

**Supports MISRA C 2012 Rule 14.4**

**9037 conditional of *#if/#elif* does not evaluate to 0 or 1**

**note** Some urge such a practice in the interest of strong typing.

**Supports MISRA C 2012 Rule 20.8**

**9038 flexible array member declared**

**note** Flexible array members can alter the behavior of `sizeof` in surprising ways. Additionally, flexible array members often require dynamic memory allocation, which may be problematic in safety critical code.

**Supports MISRA C 2012 Rule 18.7**

**9039 potentially confusing *hexadecimal/octal* escape sequence usage**

**note** An octal or hexadecimal escape sequence has been detected within a string or character literal that is not immediately followed by another escape sequence or end of literal.

**Supports MISRA C 2012 Rule 4.1**

**9040 possible struct hack declaration for *detail* member *symbol* with *integer* element(s)**

**note** Prior to C99 adding flexible array members, the now obsolete struct hack technique was widely used to create structures who's last member was an array with variable length. In addition, some compilers also allowed this

technique to create variable length array members within unions. The technique required the variable length member array(s) to be declared with a size of either 0 or 1 within the declaration of the structure or union.

```
struct S {
    int a;
    int b[0];    // warning 9040
};

union U {
    int a;
    short b[1]; // warning 9040
    char c[1];  // warning 9040
    double d;
};
```

Supports CERT C DCL38-C - Use the correct syntax when declaring a flexible array member

**9041** *goto* appears in block *string* which is not nested in block *string* which contains label *symbol*  
**note** It has been deemed safer by some experts that the block (i.e., compound statement) containing the *goto* should be the same as or nested within the block containing the label. Thus

```
{ label: { goto label; } }
```

is permitted but

```
{ goto label; { label: ; } }
```

is not. To assist the programmer, the message refers in the blocks using an identification code (e.g. "1.2.1"). This identification scheme is defined as follows:

1. The outer block has an identification of 1.
2. If a particular block is identified by *x* then its immediate subblocks, if any, are identified as *x.1*, *x.2*, *x.3*, etc.

Thus in the following 'code',

```
{ { } {{label: } { } } }
```

*label* lies in block 1.2.1.

Supports AUTOSAR17 Rule M6-6-1

Supports AUTOSAR19 Rule M6-6-1

Supports MISRA C 2012 Rule 15.3

Supports MISRA C++ Rule 6-6-1

**9042** departure from MISRA switch syntax: *detail*

**note** A *switch-statement* was found that does not comply with the MISRA *switch-statement* syntax. *detail* contains a description of the departure.

Supports AUTOSAR17 Rule M6-4-3

Supports AUTOSAR19 Rule M6-4-3

Supports MISRA C 2012 Rule 16.1

Supports MISRA C++ Rule 6-4-3

Supports MISRA C 2004 Rule 15.0

**9043** static keyword between brackets of array declaration

**note** Some advocate against using the keyword *static* in array declarations due to a perceived increased risk of

undefined behavior.

Supports MISRA C 2012 Rule 17.6

**9044 function parameter *symbol* modified**

**note** It has been advocated that function parameters be first copied to local variables where they can be modified rather than modifying the parameters directly.

Supports MISRA C 2012 Rule 17.8

**9045 complete definition of *symbol* is unnecessary in this translation unit**

**note** Some advise against including structure definitions unless the definition is required for the current module.

Supports MISRA C 2012 Dir 4.8

**9046 *symbol* is typographically ambiguous with respect to '*string*' when *detail***

**note** Some have warned against the use of identifiers that may be considered typographically ambiguous. In addition to the name of the previously seen symbol, the reasons Lint considers the identifiers to be ambiguous and the location of said previous symbol are provided in the message, if available.

Supports CERT C DCL02-C - *Use visually distinct identifiers*

Supports MISRA C 2012 Dir 4.5

**9047 FILE pointer dereferenced**

**note** At least one standards organization urges against this practice, directly or indirectly.

Supports CERT C FIO38-C - *Do not copy a FILE object*

Supports MISRA C 2012 Rule 22.5

**9048 unsigned integer literal without a 'U' suffix**

**note** An integer literal of unsigned type was found without a 'U' suffix.

Supports MISRA C 2012 Rule 7.2

Supports MISRA C 2004 Rule 10.6

**9049 increment/decrement operation combined with other operation with side-effects**

**note** An expression was seen involving an increment or decrement operator and the expression also contained potential side-effects other than those resulting from said operator. For the purpose of this message, a function call is always considered to have potential side-effects.

Supports AUTOSAR17 Rule M5-2-10

Supports AUTOSAR19 Rule M5-2-10

Supports MISRA C 2012 Rule 13.3

Supports MISRA C++ Rule 5-2-10

Supports MISRA C 2004 Rule 12.13

**9050 dependence placed on operator precedence (operators '*operator*' and '*operator*')**

**note** Reliance on operator precedence was found in a particular expression. Using parentheses, it is felt, helps clarify the order of evaluation.

Supports MISRA C 2004 Rule 12.1 (Adv)

Supports CERT C EXP00-C - *Use parentheses for precedence of operation*

Supports MISRA C 2012 Rule 12.1

Supports MISRA C 2004 Rule 12.1

Supports CWE-783 - *Operator Precedence Logic Error*

- 9051 macro '*string*' defined with the same name as a C keyword**  
**note** A macro was defined with the same name as an ISO C keyword. The use of such a macro causes undefined behavior.  
Supports MISRA C 2012 Rule 20.4
- 9052 macro '*string*' defined with the same name as a C++ keyword**  
**note** A macro was defined with the same name as an ISO C++ keyword. The use of such a macro causes undefined behavior.  
Supports MISRA C++ Rule 17-0-1
- 9053 the shift value is *negative/at least the precision of the left hand side***  
**note** A quantity with a certain *essential type*, as defined by MISRA, was shifted by a number less than zero or exceeding the number of bits used to represent that essential type.  
Supports MISRA C 2012 Rule 12.2
- 9054 designated initializer used with array of unspecified dimension**  
**note** It has been advocated, when arrays initializers contain designators, the dimension of the array should be explicitly stated in the declaration. The initializer of the array in question has been found in violation of this recommendation.  
Supports MISRA C 2012 Rule 9.5
- 9055 most closely enclosing compound statement of this '*string*' label is not a switch statement**  
**note** Labels nested inside of compound statements within the corresponding `switch` are legal but can reduce comprehension and lead to unstructured code.  
Supports AUTOSAR17 Rule M6-4-4  
Supports AUTOSAR19 Rule M6-4-4  
Supports CERT C MSC20-C - *Do not use a switch statement to transfer control into a complex block*  
Supports MISRA C 2012 Rule 16.2  
Supports MISRA C++ Rule 6-4-4  
Supports MISRA C 2004 Rule 15.1
- 9056 inline function *symbol* defined with storage-class specifier *string***  
**note** This message is issued for all inline functions defined with a storage-class specifier. `+estring` can be used to find all inline functions defined with a specific specifier. For example, `+estring(9056, extern)` will report all inline functions defined with `extern`.  
Supports MISRA C 2012 Rule 8.10
- 9057 lowercase L follows 'u' in literal suffix**  
**note** A lowercase letter "l" is used inside of a literal suffix following an upper or lowercase letter u. With some fonts, the lowercase letter "l" can be easily confused with the number one. This is less likely to happen when there is a "u" between the number and the "l" (as in 35u1), but some coding standards forbid the use of "l" in any literals. Message 620 reports the more suspicious case where the "l" immediately follows a number (as in 351).  
Supports MISRA C 2012 Rule 7.3



**9058 tag *symbol* unused outside of typedefs**

**note** A tag was used only in the course of creating a `typedef`. Was the tag unused by mistake (say a recursive reference inside the body of the struct was accidentally omitted)? Such tags are most often redundant and can be eliminated. This message is suppressed for unit checkout (`-unit_check` option).  
Supports MISRA C 2012 Rule 2.4

**9059 C comment contains C++ comment**

**note** A C++-style comment was seen inside a C-style comment. This can be confusing.  
Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*  
Supports MISRA C 2012 Rule 3.1

**9060 trigraph in comment**

**note** A trigraph was seen inside a comment. Since trigraphs are translated before preprocessing, a trigraph sequence like `??/` can have surprising results, especially in a C++ style comment where the trigraph sequence translates into a backslash.  
Supports CERT C PRE07-C - *Avoid using repeated question marks*  
Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*  
Supports MISRA C 2012 Rule 4.2

**9063 no comment or action in the 'else' that terminates 'if ... else if' chain**

**note** An `else`-branch was seen that contained neither a comment nor an actionable statement. At least one standards organization cautions against such "empty else" branches.  
Supports MISRA C 2012 Rule 15.7  
Supports MISRA C 2004 Rule 14.10  
Supports CWE-398 - *Indicator of Poor Code Quality*

**9064 goto references earlier label *symbol***

**note** A `goto` makes reference to a label appearing earlier in the code. At least one author recommends all such statements reference points later in the code in an attempt to reduce visual code complexity.  
Supports AUTOSAR17 Rule M6-6-2  
Supports AUTOSAR19 Rule M6-6-2  
Supports MISRA C 2012 Rule 15.2  
Supports MISRA C++ Rule 6-6-2

**9066 C++ comment contains C comment**

**note** A C-style comment was seen inside a C++-style comment. This can result in confusion.  
Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*  
Supports MISRA C 2012 Rule 3.1

**9067 extern array declared without size or initializer**

**note** An array was declared without a dimension. At least one standards organization advises against such a practice in the interest of safety. Note this message is not given if the array is initialized at the time of declaration.  
Supports AUTOSAR17 Rule A3-1-4  
Supports AUTOSAR19 Rule A3-1-4  
Supports MISRA C 2012 Rule 8.11  
Supports MISRA C++ Rule 3-1-3  
Supports MISRA C 2004 Rule 8.12

**9068 partial array initialization**

**note** An array has been initialized only partly. Providing an explicit initialization for each element of an array makes it clear every element has been considered. This diagnostic is not issued if the array is initialized with a `{0}` initializer or if the initializer consists entirely of designated initializers or if the array is initialized using a string literal. See also [785](#).

**Supports MISRA C 2004 Rule 9.2 (Req)**

**Supports MISRA C 2012 Rule 9.3**

**Supports MISRA C 2004 Rule 9.2**

**9069 in initializer for symbol *symbol*, initializer of type *type* needs braces or designator**

**note** In the initializer for a variable declared with aggregate (array or structure) or union type, there were insufficient braces or designators necessary to make clear which members/elements are initialized to which values. More specifically, the initializer is expected to be fully braced, e.g. with braces appearing at the beginning of every aggregate sub-object being explicitly initialized, with the following exceptions:

- If all of the initializers for a particular sub-object are designated initializers, braces are not required for that sub-object.
- String literals may be used to initialize arrays.
- An aggregate sub-object may be initialized with an object of compatible type.
- The idiomatic `{ 0 }` may be used to initialize sub-objects to an arbitrary depth without providing nested braces.

For example:

```
enum wk_type { FIRE, ICE };

struct monster {
    const char name[10];
    int hp;
    struct weakness {
        enum wk_type wk;
        double dmg_mult;
    } weak[2];
};

// Okay - all initialized sub-objects are braced, array 'name' initialized with string literal
struct monster goblin1 = {"goblin", 10, {{ICE, 2.0}, {FIRE, 1.5}}};

// 9069 - the second element of the 'weak' array is not braced
struct monster goblin2 = {"goblin", 10, {{ICE, 2.0}, FIRE, 1.5}};

// Okay - only initialized part of non-braced sub-object uses designated initializer
struct monster goblin3 = {"goblin", 20, .weak[0].wk = FIRE};

// 9069 - '1' initializes part of sub-object that is not braced
struct monster goblin4 = {"goblin", 10, .weak[0].wk = FIRE, 1};

// 9069 - initialized sub-object 'struct weakness [0]' needs additional braces
struct monster goblin5 = {"goblin", 40, {1}};
```

```
// Okay - exception for sub-objects initialized with { 0 }
struct monster goblin6 = {"goblin", 40, {0}};
```

Supports MISRA C 2012 Rule 9.2

#### 9070 function '*name*' is recursive

**note** The named function has been found to potentially call itself, either directly or indirectly. Recursion carries with it the danger of exceeding available stack space, which can lead to a run-time failure. All else being equal, the more that recursion is constrained, the easier determining the worst-case stack usage can be.

Supports AUTOSAR17 Rule A7-5-2

Supports AUTOSAR19 Rule A7-5-2

Supports CERT C MEM05-C - *Avoid large stack allocations*

Supports MISRA C 2012 Rule 17.2

Supports MISRA C++ Rule 7-5-4

Supports MISRA C 2004 Rule 16.2

#### 9071 defined macro '*name*' is reserved to the compiler

**note** A macro was defined that is reserved to the compiler. Such definition results in undefined behavior.

Supports CERT C DCL37-C - *Do not declare or define a reserved identifier*

Supports MISRA C 2012 Rule 21.1

Supports MISRA C++ Rule 17-0-1

Supports MISRA C 2004 Rule 20.1

#### 9072 parameter *integer* of function *symbol* has different name than previous declaration (*symbol* vs *symbol*)

**note**

The parameter of function *symbol* specified by *integer* has a parameter name that differs from the name of a previous declaration of the same function. Using inconsistent names within declarations of the same function can be confusing and result in misuse.

Supports AUTOSAR17 Rule M8-4-2

Supports AUTOSAR19 Rule M8-4-2

Supports MISRA C 2012 Rule 8.3

Supports MISRA C++ Rule 8-4-2

Supports MISRA C 2004 Rule 16.4

#### 9073 parameter *integer* of function *symbol* has type alias name type difference with previous declaration (*type* vs *type*)

**note**

In a function declaration or definition, the specified parameter is declared with a *type* that, while technically identical, uses a different name for the *type* than was used for the parameter in a previous declaration. For example:

```
typedef int INT;
void foo(int i);
void foo(INT i) {
    ...
}
```

would yield this message as the parameter *i* in function *foo* is declared as an *int* in both cases but in the definition the *typedef* name *INT* is used while in the preceding declaration the name *INT* is not employed. Such inconsistencies can result in unnecessary confusion.

Supports AUTOSAR17 Rule M3-9-1

Supports AUTOSAR19 Rule M3-9-1

Supports MISRA C 2012 Rule 8.3

Supports MISRA C++ Rule 3-9-1

Supports MISRA C 2004 Rule 8.3

**9074 conversion between pointer to function type *type* and differing type *type***

**note**

A conversion was seen between a pointer to function and a different type. The conversion of a pointer to a function into or from a pointer to object, pointer to incomplete type, or pointer to void results in undefined behavior and, consequently, at least one standards organization advises against such practice. This diagnostic is suppressed if the conversion is to void.

Supports CERT C DCL07-C - *Include the appropriate type information in function declarators*

Supports MISRA C 2012 Rule 11.1

**9075 external symbol *symbol* defined without a prior declaration**

**note**

If a declaration for an object is visible when that object is defined, a compiler must verify that the declaration and definition are compatible. A lack of prior declaration prevents such checking.

Supports MISRA C 2012 Rule 8.4

**9076 cast from *type* to *type* involves a pointer to an incomplete type other than void**

**note**

A conversion between two distinct types involved a pointer to an incomplete type. The resulting pointer may be improperly aligned and the result of interpreting the target memory using a different type may be a trap representation. Conversions to void and conversions to or from a pointer to void will not be reported.

Supports MISRA C 2012 Rule 11.2

**9077 missing unconditional break from final switch case**

**note**

A case at the end of a switch had no unconditional **break**. Some coding guidelines require the use of a **break** for every switch case, including the last one, for maintenance reasons. Note that this message is issued even if the case contains an unconditional **return** statement.

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.3

**9078 conversion between object pointer type *type* and integer type *type***

**note**

A conversion between a pointer type and an integer/enum type was seen. Such conversions can result in undefined behavior if the pointer value cannot be represented in the integer/enum type. This diagnostic is not given for null pointer constants.

Supports MISRA C 2012 Rule 11.4

**9079 conversion from pointer to void to other pointer type (*type*)**

**note**

Conversion of a pointer to void into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behavior.

Supports AUTOSAR17 Rule M5-2-8

Supports AUTOSAR19 Rule M5-2-8

Supports MISRA C 2012 Rule 11.5

Supports MISRA C++ Rule 5-2-8

**9080 integer null pointer constant is not the NULL macro**

**note**

An integer null pointer constant other than the NULL macro was used. Using the NULL macro makes it

clear a null pointer constant was intended.

Supports MISRA C 2012 Rule 11.9

**9081 too few independent cases for switch**

**note** A `switch` was seen with fewer than two non-consecutive case labels. A `switch` with fewer than two such cases is redundant and may indicate a programming error. See also message [9181](#).

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.6

**9082 switch should begin or end with default**

**note** Placing the `default` label either first or last makes locating it easier.

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.5

**9083 undefined macro name '*name*' is reserved to the compiler**

**note** A `#undef` was seen applied to an identifier given by *name* and that identifier is reserved to the compiler by the ISO C/C++ standards.

Supports MISRA C 2012 Rule 21.1

Supports MISRA C 2004 Rule 20.1

**9084 result of assignment operator used in *string***

**note** An assignment expression was seen inside a larger expression. The use of assignment operators, simple or compound, in combination with other arithmetic operations can significantly impair the readability of the code.

Supports AUTOSAR17 Rule M6-2-1

Supports AUTOSAR19 Rule M6-2-1

Supports MISRA C 2012 Rule 13.4

Supports MISRA C++ Rule 6-2-1

**9085 statement or comment should appear in default case**

**note** A `default` label was seen without a comment or statement between it and either the corresponding `break` or, if `default` is the last case in the `switch`, the closing `}`. Adding a statement to take action or adding a comment to explain why no action is taken is a form of defensive programming.

Supports MISRA C 2012 Rule 16.1

Supports MISRA C 2012 Rule 16.4

**9087 cast from pointer to object type (*type*) to pointer to different object type (*type*)**

**note** A cast was seen between two pointer types that differ with respect to what those types point to. Additionally, the type to which the expression was cast is not a pointer to `char`, whether signed or unsigned. At least one standards organization has cautioned against such a practice.

Supports MISRA C 2012 Rule 11.3

Supports MISRA C 2004 Rule 11.4

**9088 named signed single-bit bitfield**

**note** A named bit-field was declared with a signed data type and only one bit of width. According to the ISO C Standard, a single-bit signed bit-field has one sign bit and no value bits and, consequently does not specify a meaningful value.

Supports AUTOSAR19 Rule M9-6-4

Supports CERT C INT16-C - *Do not make assumptions about representation of signed integers*

Supports MISRA C 2012 Rule 6.2

Supports MISRA C++ Rule 9-6-4

Supports MISRA C 2004 Rule 6.5

#### 9090 switch case lacks unconditional break or throw

**note** A `switch` case was seen that did not conclude with an unconditional break. Some authors advise against such absences on the grounds they are often errors.

Supports AUTOSAR17 Rule M6-4-5

Supports AUTOSAR19 Rule M6-4-5

Supports MISRA C 2012 Rule 16.3

Supports MISRA C++ Rule 6-4-5

Supports MISRA C 2004 Rule 15.2

#### 9091 casting from pointer type *type* to integer type *type*

**note** A cast of a pointer to an integer type was seen. Since the size of the integer required when a pointer is converted to an integer is implementation defined, some coding guidelines advise against such casts.

Supports AUTOSAR17 Rule M5-2-9

Supports AUTOSAR19 Rule M5-2-9

Supports MISRA C++ Rule 5-2-9

#### 9093 the name '*name*' is reserved to the compiler

**note** A symbol was declared with a name reserved to the compiler.

Supports AUTOSAR17 Rule A17-0-1

Supports AUTOSAR17 Rule M17-0-2

Supports AUTOSAR19 Rule A17-0-1

Supports AUTOSAR19 Rule M17-0-2

Supports CERT C DCL37-C - *Do not declare or define a reserved identifier*

Supports MISRA C 2012 Rule 21.2

Supports MISRA C++ Rule 17-0-2

Supports MISRA C 2004 Rule 20.2

#### 9094 return type of function *symbol* has type alias name difference with previous declaration (*type* vs *type*)

**note** This message is similar to [9073](#) (which deals with parameter types) but applies to return types. In a declaration of a function, the return type specified, while technically identical, uses a different type name than was used for a previous declaration. For example:

```
typedef int INT;
int foo(void);
INT foo(void) {
    ...
}
```

will yield this message.

Supports AUTOSAR17 Rule M3-9-1

Supports AUTOSAR19 Rule M3-9-1

Supports MISRA C 2012 Rule 8.3

Supports MISRA C++ Rule 3-9-1

Supports MISRA C 2004 Rule 8.3

**9095 symbol *symbol* has same name as previously defined macro**

**note** A symbol was defined with the same name as a macro that was defined earlier in the same translation unit. For example:

```
#define sum(x, y) ((x)+(y))
int sum = 0;
```

will produce:

```
note 9095: symbol 'sum' has same name as previously defined macro
```

A supplemental message (891) provides the location of the offending macro definition. Note that the message is issued regardless of whether the macro definition is active at the point in which the symbol is declared. For example:

```
#define A
#undef A
int A = 0;
```

will elicit the same complaint for the declaration of A.

**Supports MISRA C 2012 Rule 5.5**

**9096 symbol *symbol* has same name as subsequently defined macro**

**note** This message is similar to 9095 but is issued for symbols defined with the same name as a macro whose definition appears *after* the declaration of the symbol. For example:

```
int A;
#define A 10
```

Unlike message 652, this message will be issued even if the macro is defined outside the scope of the symbol. For example:

```
void foo(int x) { }
#define x 10
```

will not result in a 652 warning since the definition of the x macro is outside the scope of the function parameter but 9096 will still be issued.

A supplemental message (891) provides the location of the offending macro definition.

**Supports MISRA C 2012 Rule 5.5**

**9097 unparenthesized argument to sizeof operator**

**note** An unparenthesized expression was used as the argument to the `sizeof` operator. While legal, it can result in confusion when used within a larger expression, e.g.:

```
size = sizeof x + y;
```

was this meant to be `sizeof(x) + y` or `sizeof(x + y)`? Using parenthesis can eliminate such questions.

**Supports MISRA C 2012 Rule 12.1**

**9098 pointer argument *integer* (of type *type*) to function *symbol* does not point to a pointer type or an essentially signed, unsigned, boolean, or enum type**

**note** The first or second argument to `memcpy` (or a function with semantics copied from `memcpy`) was not either 1) a pointer to a pointer or 2) a pointer to a MISRA C 2012 essentially signed, unsigned, boolean, or enum type.

**Supports MISRA C 2012 AMD1 Rule 21.16**

**9102 possible digraph sequence: '*string*'**

**note** A possible digraph was seen. At least one set of coding guidelines advises against such due to the risk of failure to meet developer expectations.

**Supports AUTOSAR17 Rule A2-6-1**

**Supports AUTOSAR19 Rule A2-5-2**

**Supports MISRA C++ Rule 2-5-1**

**9103 identifier '*name*' with static storage is reused**

**note** An identifier of the given name was seen declared static in one location and not static in another. Some coding guidelines advise against such practice due to the potential for programmer confusion.

**Supports MISRA C++ Rule 2-10-5**

**9104 octal escape sequence used**

**note** Octal escape sequences can be problematic because the inadvertent introduction of a decimal digit (i.e. 8 or 9) ends the octal escape and introduces another character. This diagnostic is not given for \0.

**Supports AUTOSAR17 Rule M2-13-2**

**Supports AUTOSAR19 Rule M2-13-2**

**Supports MISRA C++ Rule 2-13-2**

**Supports MISRA C 2004 Rule 4.1**

**Supports MISRA C 2004 Rule 7.1**

**9105 unsigned octal and hexadecimal literals require a 'U' suffix**

**note** The inclusion of such a suffix makes clear the value has unsigned type.

**Supports AUTOSAR17 Rule M2-13-3**

**Supports AUTOSAR19 Rule M2-13-3**

**Supports MISRA C++ Rule 2-13-3**

**9106 lower case literal suffix, '*string*'**

**note** Using upper case literal suffixes removes the potential for ambiguity with respect to literal values.

**Supports AUTOSAR17 Rule M2-13-4**

**Supports AUTOSAR19 Rule M2-13-4**

**Supports MISRA C++ Rule 2-13-4**

**9107 header cannot be included in more than one translation unit because of the definition of symbol *symbol***

**note** One set of guidelines advises the use of headers in such a way as to avoid the definition of objects or functions that occupy storage.

**Supports AUTOSAR17 Rule A3-1-1**

**Supports AUTOSAR19 Rule A3-1-1**

**Supports MISRA C++ Rule 3-1-1**

**Supports MISRA C 2004 Rule 8.5**

**9108 function *symbol* declared at block scope**

**note** A function was declared within the body of another function. The inner declaration will be associated with a definition in the scope enclosing the outer function for linking purposes but the name will be meaningful only



within the body of the enclosing function unless re-declared elsewhere.

If a block scope declaration occurs within a namespace in C++ then message 1798 will also be issued to highlight the heightened danger of confusion.

Supports AUTOSAR17 Rule M3-1-2

Supports AUTOSAR19 Rule M3-1-2

Supports MISRA C++ Rule 3-1-2

Supports MISRA C 2004 Rule 8.6

**9110 bit representation of a floating point type used (conversion from *type* to *type*)**

**note** The under lying bit representation of floating point values can differ from compiler to compiler, making reliance upon such representation non-portable.

Supports AUTOSAR17 Rule M3-9-3

Supports AUTOSAR19 Rule M3-9-3

Supports MISRA C++ Rule 3-9-3

Supports MISRA C 2004 Rule 12.12

**9111 boolean expression used with non-permitted operator '*string*'**

**note** The use of expressions of `bool` with certain operators, such as the bitwise operators, is not likely to be either meaningful or intended.

Supports AUTOSAR17 Rule M4-5-1

Supports AUTOSAR19 Rule M4-5-1

Supports MISRA C++ Rule 4-5-1

**9112 plain character expression used with non-permitted operator '*string*'**

**note** With the exception of the sequence of character values representing 0 thru 9, the exact value of any other particular character is not guaranteed and reliance upon such an order is non-portable.

Supports AUTOSAR17 Rule M4-5-3

Supports AUTOSAR19 Rule M4-5-3

Supports CERT C INT07-C - *Use only explicitly signed or unsigned char type for numeric values*

Supports MISRA C++ Rule 4-5-3

Supports CWE-682 - *Incorrect Calculation*

**9113 dependence placed on C++ operator precedence**

**note** The use of parentheses instead of relying upon operator precedence can help make the code easier to understand.

Supports AUTOSAR17 Rule M5-0-2

Supports AUTOSAR19 Rule M5-0-2

Supports MISRA C++ Rule 5-0-2

Supports CWE-783 - *Operator Precedence Logic Error*

**9114 implicit conversion of underlying type of integer cvalue expression from *underlying-type* to *underlying type***

**note** A prominent coding standard has defined the notion of a `cvalue` expression and, to help ensure operations in a given expression are performed within a particular fashion, the guidelines caution against such a value undergoing implicit conversions.

Supports AUTOSAR17 Rule M5-0-3

Supports AUTOSAR19 Rule M5-0-3

Supports MISRA C++ Rule 5-0-3

- 9115 implicit conversion from integer to floating point type**  
**note** Such conversions between these two types of values can result in inexact representation.  
 Supports AUTOSAR17 Rule M5-0-5  
 Supports AUTOSAR19 Rule M5-0-5  
 Supports MISRA C++ Rule 5-0-5
- 9116 implicit conversion of underlying type of floating point cvalue expression from *underlying-type* to *underlying type***  
**note** A prominent coding standard has defined the notion of a `cvalue` expression and, to help ensure operations in a given expression are performed within a particular fashion, the guidelines caution against such a value undergoing implicit conversions.  
 Supports AUTOSAR17 Rule M5-0-3  
 Supports AUTOSAR19 Rule M5-0-3  
 Supports MISRA C++ Rule 5-0-3
- 9117 implicit conversion from *underlying-type* to *underlying-type* changes signedness of underlying type**  
**note** Some such conversions can lead to implementation defined behavior. Reliance upon such behavior is, therefore, not portable.  
 Supports AUTOSAR17 Rule M5-0-4  
 Supports AUTOSAR19 Rule M5-0-4  
 Supports MISRA C++ Rule 5-0-4
- 9118 implicit conversion from floating point to integer type**  
**note** Such conversions between these two types of values can result in undefined behavior.  
 Supports AUTOSAR17 Rule M5-0-5  
 Supports AUTOSAR19 Rule M5-0-5  
 Supports CERT C FLP34-C - *Ensure that floating-point conversions are within range of the new type*  
 Supports MISRA C++ Rule 5-0-5  
 Supports CWE-197 - *Numeric Truncation Error*  
 Supports CWE-681 - *Incorrect Conversion between Numeric Types*
- 9119 implicit conversion of integer to smaller underlying type (*type* to *type*)**  
**note** A conversion was performed from an integer to a type that has a smaller MISRA C++ *underlying type*.  
 Supports AUTOSAR17 Rule M5-0-6  
 Supports AUTOSAR19 Rule M5-0-6  
 Supports MISRA C++ Rule 5-0-6
- 9120 implicit conversion of floating point to smaller underlying type (*type* to *type*)**  
**note** A conversion was performed from a floating point type to a type that has a smaller MISRA C++ *underlying type*.  
 Supports AUTOSAR17 Rule M5-0-6  
 Supports AUTOSAR19 Rule M5-0-6  
 Supports CERT C FLP03-C - *Detect and handle floating-point errors*  
 Supports MISRA C++ Rule 5-0-6  
 Supports CWE-369 - *Divide By Zero*

**9121 cast of cvalue expression from integer to floating point type****note**

A cast was used to convert a MISRA C++ cvalue expression from an integral to floating point type.

Supports AUTOSAR17 Rule M5-0-7

Supports AUTOSAR19 Rule M5-0-7

Supports MISRA C++ Rule 5-0-7

**9122 cast of cvalue expression from floating point to integer type****note**

A cast was used to convert a MISRA C++ cvalue expression from a floating point to integral type.

Supports AUTOSAR17 Rule M5-0-7

Supports AUTOSAR19 Rule M5-0-7

Supports MISRA C++ Rule 5-0-7

**9123 cast of integer cvalue expression to larger type****note**

A cast was used to convert a MISRA C++ cvalue expression of integral type to a type with a larger *underlying type*.

Supports AUTOSAR17 Rule M5-0-8

Supports AUTOSAR19 Rule M5-0-8

Supports MISRA C++ Rule 5-0-8

**9124 cast of floating point cvalue expression to larger type****note**

A cast was used to convert a MISRA C++ cvalue expression of floating point type to a type with a larger *underlying type*.

Supports AUTOSAR17 Rule M5-0-8

Supports AUTOSAR19 Rule M5-0-8

Supports MISRA C++ Rule 5-0-8

**9125 cast of integer cvalue expression changes signedness****note**

A cast was used to convert a MISRA C++ cvalue expression of integral type to an *underlying type* with a different signedness.

Supports AUTOSAR17 Rule M5-0-9

Supports AUTOSAR19 Rule M5-0-9

Supports MISRA C++ Rule 5-0-9

**9126 the result of the *operator* operator applied to an object with an underlying type of *underlying-type* must be cast to *type* in this context****note**

The ~ or << operator was applied to an operand with a MISRA C++ *underlying type* of unsigned char or unsigned short but the result was not cast to the appropriate underlying type.

Supports AUTOSAR17 Rule M5-0-10

Supports AUTOSAR19 Rule M5-0-10

Supports MISRA C++ Rule 5-0-10

**9128 plain character data mixed with non-plain-character data****note**

A prominent standard urges, since whether plain char is signed or unsigned is implementation defined, the char type not be mixed with other types.

Supports MISRA C 2004 Rule 6.2 (Req)

Supports AUTOSAR17 Rule M5-0-11

Supports AUTOSAR19 Rule M5-0-11

Supports MISRA C++ Rule 5-0-11

Supports MISRA C 2004 Rule 6.1

Supports MISRA C 2004 Rule 6.2

**9130 bitwise operator '*operator*' applied to signed underlying type**

**note** The specified bitwise operator was applied to an operand with a signed MISRA C++ *underlying type*.

Supports AUTOSAR17 Rule M5-0-21

Supports AUTOSAR19 Rule M5-0-21

Supports MISRA C++ Rule 5-0-21

**9131 *left/right* side of logical operator '*operator*' is not a postfix expression**

**note** Using only postfix-expressions with logical operators helps to improve readability of the code. Note this message is not given if the expression consists of either a sequence of only logical `&&` or a sequence of only logical `||`.

Supports AUTOSAR17 Rule M5-2-1

Supports AUTOSAR19 Rule A5-2-6

Supports MISRA C++ Rule 5-2-1

**9132 array type passed to function expecting a pointer**

**note** Array-to-pointer decay results in a loss of array bound information. A function depending upon an array to have a certain length, if that array decays to a pointer, can result in out-of-bounds operations, depending upon whether or not the bound of the original array matches with expectations.

Supports AUTOSAR17 Rule M5-2-12

Supports AUTOSAR19 Rule M5-2-12

Supports MISRA C++ Rule 5-2-12

**9133 boolean expression required for operator '*string*'**

**note** The use of non-bool operands with `!`, `&&`, or `||` is unlikely to be meaningful or intended. A more likely scenario is the programmer meant to use such an operand with one of the bitwise operators.

Supports AUTOSAR17 Rule M5-3-1

Supports AUTOSAR19 Rule M5-3-1

Supports MISRA C++ Rule 5-3-1

**9134 unary minus applied to operand with unsigned underlying type**

**note** A unary minus was applied to an expression with an unsigned MISRA C++ *underlying type*.

Supports AUTOSAR17 Rule M5-3-2

Supports AUTOSAR19 Rule M5-3-2

Supports MISRA C++ Rule 5-3-2

**9135 unary operator `&` overloaded**

**note** The unary operator `&` was overloaded.

Supports AUTOSAR17 Rule M5-3-3

Supports AUTOSAR19 Rule M5-3-3

Supports MISRA C++ Rule 5-3-3

**9136 the shift value is at least the precision of the MISRA C++ underlying type of the left hand side**

**note**

The value specified for the right hand side of a shift operator was out of bounds for the MISRA C++ *underlying type* on the left hand side of the operator.

Supports AUTOSAR17 Rule M5-8-1

Supports AUTOSAR19 Rule M5-8-1

Supports MISRA C++ Rule 5-8-1

### 9137 testing floating point values for equality

**note** This message is deprecated and will be removed in a future version. Use messages 777 and/or 9252 instead. A floating point value was tested, directly or indirectly, for (in)equality with another value.

### 9138 null statement not on line by itself

**note** A null statement was encountered that, before preprocessing, did not appear on a line by itself. Comments following the null statement are allowed as long as there is whitespace separating the null statement from the comment.

Supports AUTOSAR17 Rule M6-2-3

Supports AUTOSAR19 Rule M6-2-3

Supports MISRA C++ Rule 6-2-3

Supports MISRA C 2004 Rule 14.3

Supports CWE-398 - *Indicator of Poor Code Quality*

### 9139 case label follows default in switch statement

**note** A case label was encountered following the default label of a switch statement.

Supports AUTOSAR17 Rule M6-4-6

Supports AUTOSAR19 Rule M6-4-6

Supports MISRA C++ Rule 6-4-6

Supports MISRA C 2004 Rule 15.3

### 9141 global declaration of symbol *symbol*

**note** The specified symbol was declared in the global namespace.

Supports AUTOSAR17 Rule M7-3-1

Supports AUTOSAR19 Rule M7-3-1

Supports MISRA C++ Rule 7-3-1

### 9142 function main declared outside the global namespace

**note** A function with the name 'main' was declared that was not the global main function.

Supports AUTOSAR17 Rule M7-3-2

Supports AUTOSAR19 Rule M7-3-2

Supports MISRA C++ Rule 7-3-2

### 9144 using directive used: '*string*'

**note** A using directive was encountered.

Supports AUTOSAR17 Rule M7-3-4

Supports AUTOSAR19 Rule M7-3-4

Supports MISRA C++ Rule 7-3-4

### 9145 using *declaration/directive* in header '*file*'

**note** A using directive or using declaration was encountered in a header file. This message is not issued for using

declarations in class or function scope.

Supports AUTOSAR17 Rule M7-3-6

Supports AUTOSAR19 Rule M7-3-6

Supports MISRA C++ Rule 7-3-6

#### 9146 multiple declarators in a declaration

**note** A declaration was encountered that contains multiple declarators. For example:

```
int i, j;
```

will elicit this message.

Supports AUTOSAR17 Rule A7-1-7

Supports AUTOSAR17 Rule M8-0-1

Supports AUTOSAR19 Rule A7-1-7

Supports AUTOSAR19 Rule M8-0-1

Supports CERT C DCL04-C - *Do not declare more than one variable per declaration*

Supports MISRA C++ Rule 8-0-1

#### 9147 implicit function-to-pointer decay

**note** The unadorned name of a function was encountered that was not part of a function call.

Supports AUTOSAR17 Rule M8-4-4

Supports AUTOSAR19 Rule M8-4-4

Supports MISRA C 2012 AMD3 Rule 17.12

Supports MISRA C++ Rule 8-4-4

Supports MISRA C 2004 Rule 16.9

#### 9148 '=' should initialize either all enum members or only the first for enumerator *symbol*

**note** Unintentional duplication of enumerator values can occur when an enumeration consists of members with explicit and implicit values.

Supports AUTOSAR17 Rule A7-2-4

Supports AUTOSAR19 Rule A7-2-4

Supports CERT C INT09-C - *Ensure enumeration constants map to unique values*

Supports MISRA C++ Rule 8-5-3

Supports MISRA C 2004 Rule 9.3

#### 9149 bit field must be explicitly signed integer, unsigned integer, or bool

**note** When using 'int' or 'wchar\_t' as the bit-field type, it is implementation defined whether or not the type used is a signed type. Explicitly specifying 'signed' or 'unsigned' makes it clear what type will be used as the underlying type.

Supports MISRA C 2012 Rule 6.1

Supports MISRA C++ Rule 9-6-2

Supports MISRA C++ Rule 9-6-3

#### 9150 non-private data member *symbol* within a non-POD structure

**note** A member of a non-POD structure was declared public or protected.

Supports AUTOSAR17 Rule M11-0-1

Supports AUTOSAR19 Rule M11-0-1

Supports MISRA C++ Rule 11-0-1

- 9151 abstract class *symbol* declares public copy assignment operator *symbol***  
**note** A public copy assignment operator was declared in an abstract class.  
 Supports MISRA C++ Rule 12-8-2
- 9153 viable set contains both function *symbol* and template *symbol***  
**note** In a context where a name resolves either to a non-template function or to a specialization of a function template (typically a call), the set of viable candidates included both.  
 Supports AUTOSAR17 Rule A14-8-1  
 Supports MISRA C++ Rule 14-8-2
- 9154 throwing a pointer**  
**note** A pointer type was passed to a `throw` expression. It may not be clear who is responsible for cleaning up the pointed to object.  
 Supports AUTOSAR17 Rule A15-1-2  
 Supports AUTOSAR19 Rule A15-1-2  
 Supports MISRA C++ Rule 15-0-2
- 9156 rethrow outside of catch block will call `std::terminate` if no exception is being handled**  
**note** An empty throw expression was encountered outside of a try-catch block. An empty throw re-throws the currently handled exception. If there is no such exception `std::terminate()` will be called. This is likely to be unintended.  
 Supports AUTOSAR17 Rule M15-1-3  
 Supports AUTOSAR17 Rule A15-5-2  
 Supports AUTOSAR17 Rule A15-5-3  
 Supports AUTOSAR19 Rule M15-1-3  
 Supports AUTOSAR19 Rule A15-5-2  
 Supports AUTOSAR19 Rule A15-5-3  
 Supports MISRA C++ Rule 15-1-3
- 9158 `#define` used within *string symbol* for macro '*name*'**  
**note** A macro was defined inside the braced region of the entity described by *string* (such as `function` or `class`). Such usage could imply the belief that the scope of the macro definition is limited to the braced region, which is not the case.  
 Supports AUTOSAR17 Rule M16-0-2  
 Supports AUTOSAR19 Rule M16-0-2  
 Supports MISRA C++ Rule 16-0-2  
 Supports MISRA C 2004 Rule 19.5
- 9159 `#undef` used within *string symbol* for macro '*name*'**  
**note** A macro was undefined inside the braced region of the entity described by *string* (such as `function` or `class`). Such usage could imply the belief that the scope of the directive is limited to the braced region, which is not the case.  
 Supports AUTOSAR17 Rule M16-0-2  
 Supports AUTOSAR19 Rule M16-0-2  
 Supports MISRA C++ Rule 16-0-2  
 Supports MISRA C 2004 Rule 19.5

**9160 unknown preprocessor directive '*string*' in conditionally excluded region**

**note** Within a conditionally excluded region, a line that started with a `#` was seen but was not part of a valid preprocessing directive. Error 16 is produced if an unknown preprocessor directive appears in a non-excluded region.

Supports AUTOSAR17 Rule M16-0-8

Supports AUTOSAR19 Rule M16-0-8

Supports MISRA C 2012 Rule 20.13

Supports MISRA C++ Rule 16-0-8

Supports MISRA C 2004 Rule 19.16

**9162 use of '*string*' at global scope**

**note** Either a `static_assert()` or a using-declaration was seen at global scope, as indicated by the *string*.

Supports AUTOSAR17 Rule M7-3-1

Supports AUTOSAR19 Rule M7-3-1

Supports MISRA C++ Rule 7-3-1

**9165 function *symbol* defined with a variable number of arguments**

**note** The named function is defined to take a variable number of arguments. At least one author advises against such a practice because doing so avoids the type checking provided by the compiler.

Supports AUTOSAR17 Rule A8-4-1

Supports AUTOSAR19 Rule A8-4-1

Supports MISRA C++ Rule 8-4-1

Supports MISRA C 2004 Rule 16.1

**9167 macro '*name*' defined in *string symbol* not undefined in same *string***

**note** A macro was defined inside of a declaration of a function, class/struct/union, namespace, or enumeration and was not undefined within the braced region of that declaration. The macro will persist beyond the end of the declaration, which may not be intended. For example:

```
void foo() {
    #define A ...

}
```

will result in the message:

```
macro 'A' defined in function 'foo' not undefined in same function
```

**9168 variable *symbol* has type alias name difference with previous declaration (*type* vs *type*)**

**note** A variable is declared in two places with types that, while technically identical, have different alias names. For example:

```
typedef int INT;
extern int var;
INT var;           // note 9168
```

Supports AUTOSAR17 Rule M3-9-1

Supports AUTOSAR19 Rule M3-9-1

Supports MISRA C++ Rule 3-9-1

**9169 constructor *symbol* can be used for implicit conversions from fundamental type *type***

**note** A constructor was found that could be used for implicit conversions from a fundamental type. This message



is similar to 1931 but only reports instances where the implicit conversion is from a fundamental type (e.g. integer and floating point types but not pointers, references, arrays, classes, etc.). Like message 1931, if the constructor is declared with the keyword **explicit**, this message will not be emitted. This message is also not be emitted for variadic constructors.

Supports AUTOSAR17 Rule A12-1-4

Supports AUTOSAR19 Rule A12-1-4

Supports MISRA C++ Rule 12-1-3

**9170 pure function *symbol* overrides non-pure function *symbol***

**note** The specified function is declared as pure but overrides a non-pure function in a base class. Was this a mistake?

Supports AUTOSAR17 Rule M10-3-3

Supports AUTOSAR19 Rule M10-3-3

Supports MISRA C++ Rule 10-3-3

**9171 downcast of polymorphic type *type* to type *type***

**note** A cast was used to convert a pointer to a polymorphic type (a class that contains or inherits one or more virtual functions) to a pointer to a derived class.

Supports AUTOSAR17 Rule M5-2-3

Supports AUTOSAR19 Rule M5-2-3

Supports MISRA C++ Rule 5-2-3

**9172 bitwise operator '*operator*' used with non-constant operands of differing underlying types (*underlying-type* and *underlying-type*)**

A bitwise operator was used whose operands did not have the same MISRA C++ *underlying type*. This message is not produced if either operand is an integer constant expression.

Supports AUTOSAR17 Rule M5-0-20

Supports AUTOSAR19 Rule M5-0-20

Supports MISRA C++ Rule 5-0-20

**9173 use of non-placement allocation function *symbol***

**note** The use of **new** or **delete** was encountered that will allocate or deallocate dynamic memory. Placement **new** is not reported as it does not allocate memory.

Supports MISRA C++ Rule 18-4-1

**9174 *type* is a virtual base class of *symbol***

**note** A class derivation was marked as virtual; some coding standards prohibit virtual inheritance due to the potential complexities involved.

Supports AUTOSAR17 Rule M10-1-1

Supports AUTOSAR19 Rule M10-1-1

Supports MISRA C++ Rule 10-1-1

**9175 function *symbol* has void return type and no external side-effects**

**note** The specified function does not appear to have any external side-effects and does not return any information so what is the purpose of calling the function?

Supports AUTOSAR17 Rule M0-1-8

Supports AUTOSAR19 Rule M0-1-8

Supports MISRA C++ Rule 0-1-8

**9176 pointer type *type* converted to unrelated pointer type *type***

**note** A pointer was converted (implicitly or explicitly) to a different pointer type and the source pointee type was not a class or structure derived from the destination pointee type.

**Supports MISRA C++ Rule 5-2-7**

**9177 condition of '*string*' statement has non-Boolean type *type***

**note** The controlling expression of a `do`, `for`, `if`, or `while` statement had a non-Boolean type before applying the contextual Boolean conversion that occurs for the control statement. For example:

```
void foo(int value) {
    if (value) { ... }           // Note 9177
    if ((bool)value) { ... }     // OK
    if (value != 0) { ... }      // OK
}
```

As indicated in the above example, a cast to `bool` can be used to make the intention explicit and suppress this message. 9177 is not issued if the condition contains a variable declaration, e.g.:

```
int bar();
void foo() {
    if (int value = bar()) { ... } // OK
}
```

If a user-defined conversion operator is employed to perform implicit Boolean conversion of the controlling expression, the message is issued only if the corresponding conversion operator is not declared with the `explicit` keyword. For example:

```
struct X1 { operator bool(); };
struct X2 { explicit operator bool(); };
void foo(X1 x1, X2 x2) {
    if (x1) { ... }           // Note 9177
    if (x2) { ... }           // OK, explicit conversion operator
}
```

The *string* parameter indicates the type of statement for which the non-Boolean condition appeared and is one of "do", "for", "if", or "while".

**Supports AUTOSAR17 Rule A5-0-2**

**Supports AUTOSAR19 Rule A5-0-2**

**Supports MISRA C++ Rule 5-0-13**

**9178 predicate of conditional operator has non-Boolean type *type***

**note** The predicate of a conditional operator had a non-Boolean type before applying the implicit Boolean conversion. For example:

```
void foo(int value) {
    value ? x() : y();           // Note 9178
    (bool)value ? x() : y();     // OK
    (value != 0) ? x() : y();    // OK
}
```

As indicated in the above example, a cast to `bool` will suppress this message.

If a user-defined conversion operator is employed to perform implicit Boolean conversion of the predicate, the message is issued only if the corresponding conversion operator is not declared with the `explicit` keyword.

**Supports AUTOSAR17 Rule M5-0-14**

Supports AUTOSAR19 Rule M5-0-14

Supports MISRA C++ Rule 5-0-14

**9181 switch contains fewer than two non-default switch cases**

**note** A `switch` was seen with fewer than two non-default cases. A `switch` with fewer than two cases might be better expressed as an `if` statement. See also message [9081](#).

Supports AUTOSAR17 Rule A6-4-1

Supports AUTOSAR19 Rule A6-4-1

**9183 qualifier '*string*' precedes typedef type *type***

**note** This message is issued when the `const` or `volatile` qualifiers appear before a typedef or `using` name in a type specifier sequence. Placing `const` or `volatile` before a typedef or `using` name can provide a false impression of how the qualifier affects the type. For example:

```
typedef int * INTPTR;
volatile INTPTR counter1; // Note 9183
INTPTR volatile counter2; // OK
```

The declaration of `counter1` may lead to the incorrect impression that its type is "pointer to `volatile int`" instead of its actual type of "volatile pointer to *non-volatile int*". Analogous issues exist for the `const` qualifier. Placing the qualifier after the typedef name doesn't change the semantic meaning but reduces the chance of misinterpreting the impact of the qualifier. The *string* parameter is `const` or `volatile` and the *type* parameter contains the typedef type that follows the qualifier.

Supports AUTOSAR17 Rule A7-1-3

Supports AUTOSAR19 Rule A7-1-3

**9185 assignment operator *symbol* declared without lvalue ref-qualifier**

**note** A member assignment operator function was declared without an lvalue reference qualifier, the presence of which would limit assignment to lvalue objects. Allowing assignment to rvalue class objects can result in subtle logic errors and may be confusing since assignment to rvalues of fundamental types is not possible using built-in assignment operators. Assignment operators include copy assignment, move assignment, simple assignment, and compound assignment (e.g. `operator+=`). A member function is declared with an lvalue reference qualifier by adding `&` after the function's parameter list. For example:

```
class X {
    X& operator+=(int);    // Message 9185
    X& operator-=(int) &;  // OK
}
```

Supports AUTOSAR17 Rule A12-8-7

Supports AUTOSAR19 Rule A12-8-7

**9186 non-Boolean return type *type* for comparison function *symbol***

**note** An overloaded comparison operator function (one of `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==`, or `operator!=`) was declared to return a non-dependent type other than `bool`. The built-in versions of these operators that operate on fundamental types, as well as the overloaded versions defined in the C++ Standard library, return `bool` so declaring a comparison operator with a different return type implies unconventional semantics which may be confusing.

Supports AUTOSAR17 Rule A13-2-3

Supports AUTOSAR19 Rule A13-2-3

- 9187 non-const overloaded subscript operator *symbol* declared without a corresponding const version**  
**note** A non-**const** member function or member function template overloads the subscript operator and there is no corresponding **const** version of the function declared in the same class. It is sometimes advised that a **const** version of the subscript operator be provided with a non-**const** version. The *corresponding const* version is expected to have the same parameter type, ref-qualifier, and volatile qualifier as the non-**const** version. The implemented semantics of the subscript operator for some classes, such as `std::map` (which adds the element referenced by the subscript operator if it does not exist), may not lend itself to a **const** version. The *symbol* parameter contains the offending function and can be used to suppress the message on a case-by-case basis using the `-esym` option. The option `-esym(9187,std:*)` can be used to suppress the message for classes defined in the Standard C++ library.  
**Supports AUTOSAR17 Rule A13-5-1**  
**Supports AUTOSAR19 Rule A13-5-1**
- 9202 bitfield *symbol* declared as member of union *symbol***  
**note** A bitfield was declared as the member of a union. This is unusual as bitfields are typically used within structures to save space but all members of a union occupy the same space so that benefit does not apply. Bitfields are also not useful for type punning purposes as the location of the bits corresponding to the bitfield are implementation defined.  
**Supports MISRA C 2012 AMD3 Rule 6.3**
- 9203 declaration of *symbol* contains an alignment attribute whose expression evaluates to zero**  
**note** An alignment attribute (such as `alignas` or `_Alignas`) was encountered in the declaration of the specified symbol with an integer constant expression that evaluated to a zero value. While valid, such attributes are ignored so their presence is suspicious. If the expression is the result of a macro expansion, this message may be suppressed using `-emacro`.  
**Supports MISRA C 2012 AMD3 Rule 8.16**
- 9204 hexadecimal escape sequence used**  
**note** A hexadecimal escape sequence (`\x`) was used inside a character or string literal.  
**Supports MISRA C 2004 Rule 4.1**
- 9205 declaration of *symbol* contains multiple alignment attributes**  
**note** Multiple alignment attributes were encountered in the declaration of the specified symbol. When multiple alignment attributes are provided, all but the strictest one is ignored rendering the other attributes superfluous.  
**Supports MISRA C 2012 AMD3 Rule 8.17**
- 9208 generic selection does not contain any non-default associations**  
**note** A generic selection was encountered whose only provided association was a default association. Generic selections provide a mechanism to produce a value that is dependent on the type of a controlling expression. A generic selection that only contains a default association will always produce a value that is independent of its controlling expression making its use superfluous.  
**Supports MISRA C 2012 AMD3 Rule 23.3**
- 9209 plain character data used with prohibited operator *string***  
**note** The plain `char` type is defined by the implementation to have the same size and range as either `signed char` or `unsigned char` but is a separate and distinct type. For this reason, it is often recommended that `char` be used for character data and `signed char` and `unsigned char` be used for numeric data. This message reports when an object of plain `char` type is used as an operand to a unary operator or a binary operator

other than `=`, `==`, and `!=`.

**Supports CERT C STR09-C** - *Don't assume numeric values for expressions with type plain character*

**Supports MISRA C 2004 Rule 6.1**

**9210** **default association does not appear as either the first or the last association of generic selection**  
**note** A generic selection was encountered with a default association that does not appear as either the first or last in the association list. Placing default associations at the beginning or end of the association list helps them to stand out and clarifies the intent of the selection structure.  
**Supports MISRA C 2012 AMD3 Rule 23.8**

**9211** **generic selection is not expanded from a function-like macro**  
**note** A generic selection was encountered which was not expanded from a function-like macro or whose controlling expression is not the result of a macro argument's expansion. When a generic selection is employed outside of a macro expansion, the type of the controlling expression is predetermined so querying it is not necessary.  
**Supports MISRA C 2012 AMD3 Rule 23.1**

**9212** **bit field type *type* is not explicitly signed int or unsigned int**  
**note** A bit field was defined with a type other than `signed int` or `unsigned int` or with a `typedef` that is defined using one of these two explicit types. When using plain `int` as a bit-field type, the signedness of the type used is implementation defined. Only `int` (signed and unsigned) and `_Bool` (in C99) are sanctioned for use in bit-fields, use of any other type results in implementation-defined behavior. `-etype(9212, _Bool)` can be used to suppress this message for the C99 `_Bool` type. See also message [9149](#), which is similar but more lenient.  
**Supports MISRA C 2004 Rule 6.4**

**9213** **controlling expression contains a call to function *symbol* which will not be called**  
**note** The controlling expression of a *generic-selection* contains a function call. The controlling expression is only used for its type and therefore, the function will not be called. The related message [2419](#) will report controlling expressions containing a side effect. For example:

```
int x = 0;
int foo() { return 0; }
_Generic(x++, int: 1, default: 0); // Warning 2419
_Generic(foo(), int: 1, default: 0); // Note 9213
```

This message is not emitted when the controlling expression is expanded from a macro argument.

**Supports MISRA C 2012 AMD3 Rule 23.2**

**9214** **default association will be selected because *type* is not implicitly converted to *type***  
**note** A *generic-selection* was encountered that selected a default association when an implicit pointer conversion might have been expected. Unlike arguments passed to a function, the type of the controlling expression is not implicitly converted to match an association. Since qualifiers must match exactly, selecting the default association might be surprising behavior. For example:

```
char* S = "Hello!";
void foo(const char* str);
foo(S); // OK
_Generic(S, const char*: 1, default: 0); // Note 9214, selects default
```

This message is only issued when the type of the controlling expression is a pointer or null pointer constant.

**Supports MISRA C 2012 AMD3 Rule 23.5**

**9215 unnamed parameter for 'virtual/non-virtual' function *symbol***

**note** This message is emitted when an unnamed function parameter is encountered in the definition of a function.

Supports AUTOSAR17 Rule M0-1-11

Supports MISRA C++ Rule 0-1-11

Supports MISRA C++ Rule 0-1-12

**9216 essential type of the controlling expression '*string*' does not match its standard type *type***

**note** The selected association of a generic selection is not consistent with the MISRA C 2012 essential type of the controlling expression. This message is not applied when the controlling expression is an integer constant expression that is neither a character constant nor boolean. For example:

```
short s = 0;
_Generic(s + s, int: 0, short: 1); // Note 9216 (selects int, essentially short)
_Generic('c', int: 0, char: 1);    // Note 9216 (selects int, essentially char)
_Generic(25 + 35, int: 0, short: 1); // OK, integer constant expression
```

This message is not emitted in C++.

Supports MISRA C 2012 AMD3 Rule 23.6

**9218 arguments to type-generic macro '*macro*' resulting in call to function *symbol* have different types *type* and *type***

**note** The operand arguments passed to a multi-argument type-generic macro defined in `tgmath.h` have different standard types. All arguments to such macros should have the same standard type, after integer promotion has been applied to any integer arguments. This generally clarifies the relationship between input and outputs as well as handling platform specific cases where the deduced common real type may be an unexpected size or lose precision.

```
float f;
double d;
pow(f, d); // Note 9218
pow(f, f); // No message issued
```

Supports MISRA C 2012 AMD3 Rule 21.23

**9219 parameter *integer* of generic selection macro '*macro*' is not evaluated exactly once**

**note** The macro argument appearing in the controlling expression of an expanded *generic-selection* is not evaluated exactly once. The arguments in a controlling expression are only used for their type and are never evaluated. The macro argument should be expanded in each of the result expressions or elsewhere in the macro body, and the number of times it is evaluated should not depend on which association is selected. An exception is made if all of the result expressions are constant expressions and the macro argument never expands outside of the controlling expression. For example:

```
#define gen1(X) _Generic((X), float: funcf(X), default: 0) // Note 9219
#define gen2(X) (_Generic((X), float: funcf, default: func1) (X)) // OK
#define gen3(X) _Generic((X), float: 1, default: 0) // OK, exception
```

This message does not depend on the presence of side effects in the operand expression.

Supports MISRA C 2012 AMD3 Rule 23.7

**9224 expression is not effectively boolean and must be explicitly tested for zero**

**note** An expression that is not "effectively Boolean" is being implicitly tested for zero. For example, given that `x` is an `int`:

```
if (x)
```

will elicit this message while:

```
if (x != 0)
```

will not.

"Effectively Boolean" values are produced by the operators `==`, `!=`, `<=`, `>=`, `<`, `>`, `!`, `||`, and `&&`. An expression will also be considered "effectively Boolean" if the [Strong Type](#) of the expression is Boolean.

Supports MISRA C 2004 Rule 13.2

**9225** integral expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is not a wider integer type of the same signedness

note

An integral expression was implicitly converted to a *type* that was not a wider *type* of the same signedness.

Supports MISRA C 2004 Rule 10.1

**9226** integral expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is *string*

note

A complex integral expression was implicitly converted to a different *type* or a non-constant integral expression was implicitly converted to a different *type* while being passed to or returned from a function. "Complex" here means an expression that is not an lvalue and is not a function return value.

Supports MISRA C 2004 Rule 10.1

**9227** floating expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is not a wider floating type

note

A floating point expression was implicitly converted to a *type* that is not a wider *type*.

Supports MISRA C 2004 Rule 10.2 (Req)

Supports CERT C FLP03-C - Detect and handle floating-point errors

Supports CERT C FLP34-C - Ensure that floating-point conversions are within range of the new type

Supports MISRA C 2004 Rule 10.2

Supports CWE-197 - Numeric Truncation Error

Supports CWE-369 - Divide By Zero

Supports CWE-681 - Incorrect Conversion between Numeric Types

**9228** floating expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is *string*

note

A floating point expression was implicitly converted to a different type in a context in which a cast should be used to be compliant with MISRA C 2004. The context is provided in *string*, which is one of a complex expression, a function argument, or a return value.

Supports MISRA C 2004 Rule 10.2 (Req)

Supports MISRA C 2004 Rule 10.2

**9229** complex integral expression may only be cast to another integral type of the same signedness no wider than the original type

note

A complex expression with integral type was cast to a type with different signedness or whose *underlying type* is wider than the *underlying type* of the expression.

Supports MISRA C 2004 Rule 10.3

- 9230** **complex floating expression may only be cast to another floating type no wider than the original type**  
*note* A complex expression with floating point type was cast to a type with whose *underlying type* is wider than the *underlying type* of the expression.  
  
Supports MISRA C 2004 Rule 10.4
- 9231** **result of *operator* applied to operand of type *type* must be immediately cast to *type***  
*note* The ~ or << operator was applied to an operand with a MISRA C *underlying type* of unsigned char or unsigned short but the result was not cast to the appropriate underlying type.  
Supports MISRA C 2004 Rule 10.5
- 9232** **expected/did not expect an effectively boolean argument for operator *operator***  
*note* An "effectively Boolean" expression was used as an operand to an operator that should not operate on such an expression, or an operator for which an "effectively Boolean" expression was expected was not provided one. Specifically, the operators &&, ||, and ! should contain only "effectively Boolean" operands and "effectively Boolean" operands should not be used with operators other than &&, ||, !, =, ==, !=, and ?:. See the description of [9224](#) for more information on what is considered "effectively Boolean".  
Supports MISRA C 2004 Rule 12.6
- 9233** **bitwise operator *operator* may not be applied to operand with signed underlying type**  
*note* An expression with a MISRA C signed *underlying type* was provided as an operand to a bitwise operator.  
Supports CERT C INT13-C - Use bitwise operators only on unsigned operands  
Supports MISRA C 2004 Rule 12.7  
Supports CWE-682 - Incorrect Calculation
- 9234** **shift amount exceeds size of operand's underlying type**  
*note* An expression was shifted by a negative amount or an amount greater than the bit width of the expression's MISRA C *underlying type*.  
Supports MISRA C 2004 Rule 12.8
- 9235** **unary minus applied to operand with unsigned underlying type**  
*note* The unary minus operator was applied to an expression with an unsigned MISRA C *underlying type*.  
Supports MISRA C 2004 Rule 12.9
- 9236** **assignment operator may not be used within a boolean-valued expression**  
*note* An assignment operator was used within a Boolean context, such as comparing the result of assignment to a specific value.  
Supports MISRA C 2004 Rule 13.1
- 9237** **conversion between pointer to function type *type* and differing non-integral type *type***  
*note* A conversion was performed between a pointer to function and a pointer to a different type that was not a pointer to an integral type.  
Supports MISRA C 2004 Rule 11.1
- 9238** **switch condition may not be boolean**  
*note*



The conditional expression of a `switch` statement has an "effectively Boolean" type. See the description of [9224](#) for more information on what is considered "effectively Boolean".

Supports MISRA C 2004 Rule 15.4

**9240** *left/right* side of logical operator '*operator*' is not a primary expression

**note** This message is issued when the operands of the `||` and `&&` operators are not *primary-expressions*.

Supports MISRA C 2004 Rule 12.5

**9252** testing floating point for equality using exact value

**note** Message [777](#) is issued when an object with a floating point type is tested for equality using either `==` or `!=`. Message [777](#) is not issued when one of the operands is a value that can be represented exactly in the corresponding floating point representation, such as 0 or 13.5. In such cases, this message is issued instead.

Supports AUTOSAR17 Rule M6-2-2

Supports AUTOSAR19 Rule M6-2-2

Supports CERT C FLP00-C - *Understand the limitations of floating-point numbers*

Supports CERT C FLP02-C - *Avoid using floating-point numbers when precise computation is needed*

Supports MISRA C++ Rule 6-2-2

Supports MISRA C 2004 Rule 13.3

**9254** continue statement encountered

**note** A `continue` statement was seen. Some coding guidelines forbid the use of `continue` statements.

Supports MISRA C 2004 Rule 14.5

**9259** C comment contains '://' sequence

**note** Message [9059](#) reports on cases where a C comment contains what may be a C++ comment, e.g. the sequence `'//'`. Because including URLs inside of comments is a common practice, message [9059](#) is not issued when the `'//'` sequence is immediately preceded by a `'.'` to prevent the message from being issued in cases such as:

```
/* See https://example.com for details */
```

This message fills the gap by reporting on the instances not reported by [9059](#).

Supports CERT C MSC04-C - *Use comments consistently and in a readable fashion*

**9260** C++ style comment used

**note** A C++-style comment (`//`) was encountered. Such comments were not part of C until C99 and may not be consistently supported by older compilers.

Supports MISRA C 2004 Rule 2.2

**9264** array subscript applied to variable *symbol* declared with non-array type *type*

**note** The base of an array subscript operation was not declared as an array (i.e., it was declared as a pointer). Some coding guidelines suggest that array subscript operations should only be applied to array types.

Supports MISRA C 2004 Rule 17.4

**9272** parameter *integer* of function *symbol* has different name than overridden function *symbol* (*symbol* vs *symbol*)

**note** This message is similar to [9072](#) but applied to differences between overridden functions. In the declaration of a function, the *name* given to the specified parameter is different from the *name* given for the same parameter in the declaration of one of the functions being overridden. For example:

```

    struct A {
        virtual void foo(int width);
    };
    struct B : A {
        void foo(int depth);
    };

```

will yield this message because `A::foo` uses `width` as the name of the first parameter while the overridden function `B::foo` uses the name `depth`.

Supports AUTOSAR17 Rule M8-4-2

Supports AUTOSAR19 Rule M8-4-2

Supports MISRA C++ Rule 8-4-2

**9273** parameter *integer* of function *symbol* has type alias name difference with overridden function *symbol* (*type* vs *type*)

note

This message is similar to [9073](#) but applies to differences between overridden functions. In the declaration of a function, the specified parameter is declared with a *type* that, while technically identical, uses a different *name* for the *type* than was used for the parameter in the declaration of one of the functions that this function overrides. For example:

```

    typedef int INT;

    struct A {
        virtual void foo(int);
    };

    struct B : A {
        void foo(INT);
    };

```

will yield this message because `A::foo` is declared with a first parameter of type `int` while the overridden function `B::foo` declared the parameter with type `INT`, a type alias name difference.

**9287** cast from pointer to object type (*type*) to pointer to char type (*type*)

note

A cast was performed between a pointer to an object type and a pointer to a character type. The actual pointer types are provided in the message.

Supports MISRA C 2004 Rule 11.4

**9288** unnamed signed single-bit bitfield

note

An unnamed signed bit-field was declared with a single bit.

Supports MISRA C 2004 Rule 6.5

**9294** return type of function *symbol* has type alias name difference with overridden function *symbol* (*type* vs *type*)

note

This message is similar to [9094](#) but applies to differences between overridden functions. In a declaration of a function, the return type specified, while technically identical, uses a different type name than was used for the declaration of one of the functions that this function overrides. For example:

```

    typedef int INT;

    struct A {
        virtual int foo();
    };

```

```
};

struct B : A {
    INT foo();
};
```

will yield this message.

**9295 conversion between object pointer type *type* and non-integer arithmetic essential type 'essential-type'**

This message is issued when a cast is used to convert between a pointer to object type and a non-integer arithmetic MISRA C 2012 essential type. For example:

```
enum color { RED, GREEN, BLUE };
void foo(int *pi) {
    enum color c = (enum color)pi; // Note 9295
}
```

Supports MISRA C 2012 Rule 11.7

**9401 function *symbol* returns pointer to void**

**note** The specified function returns a pointer to void, which some consider to be unsafe because it can compromise type safety.

**9402 function *symbol* parameter *integer* is void pointer**

**note** The specified function accepts a void pointer as an argument, which some consider to be unsafe because such pointers can compromise type safety.

**9403 function *symbol* parameter *integer* has same unqualified type (*type*) as previous parameter**

**note** A function has two consecutive parameters of the same (unqualified) type. Functions that accept many arguments can be difficult to use correctly as the chance of misordered arguments increases as the number of parameters increases. When arguments are of different types, misordered arguments are more likely to be detected by the compiler. When consecutive parameters are of the same type, calls to the function that accidentally transpose the arguments are less likely to be noticed.

**9404 destructor for class *symbol* should be declared 'noexcept'**

**note** Given that destructors should never `throw`, declaring them as 'noexcept' is wise as it allows the compiler to ensure this is the case.

**9405 move constructor for class *symbol* should be declared 'noexcept'**

**note** Move constructors should not `throw`; declaring them as 'noexcept' allows the compiler to ensure this is the case.

**9406 move assignment operator *symbol* should be declared 'noexcept'**

**note** Move assignment functions should not `throw`; declaring them as 'noexcept' allows the compiler to ensure this is the case.

- 9407 copy assignment operator *symbol* should not be virtual**  
**note** A copy assignment operator was declared as `virtual`; this is rarely the right thing to do. Virtual move assignment operators are reported by [9410](#). Virtual non-copy, non-move assignment operators are reported by [9438](#).  
**Supports AUTOSAR17 Rule A10-3-5**  
**Supports AUTOSAR19 Rule A10-3-5**
- 9408 copy assignment operator *symbol* should take a `const` reference type**  
**note** A copy assignment operator should take a `const` reference argument.
- 9409 copy assignment operator *symbol* should return a non-const lvalue-reference type**  
**note** A copy assignment operator should return a non-const lvalue-reference type.
- 9410 move assignment operator *symbol* should not be virtual**  
**note** A move assignment operator was declared as `virtual`; this is rarely the right thing to do. Virtual copy assignment operators are reported by [9407](#). Virtual non-copy, non-move assignment operators are reported by [9438](#).  
**Supports AUTOSAR17 Rule A10-3-5**  
**Supports AUTOSAR19 Rule A10-3-5**
- 9411 move assignment operator *symbol* should take a non-const reference type**  
**note** A move assignment operator was declared whose argument is not a non-const reference.
- 9412 move assignment operator *symbol* should return a non-const lvalue-reference type**  
**note** A move assignment operator was declared that doesn't return a non-const lvalue-reference type.
- 9413 class *symbol* contains data members of differing access levels**  
**note** A class contains data members declared with different access levels.
- 9414 'typeid' used on expression with side effect**  
**note** If the operand of the `typeid` operator is an expression, it is not usually evaluated. Using `typeid` with an expression that would have side effects if it were evaluated could result in confusion about whether the apparent side-effects will actually take place.  
**Supports AUTOSAR17 Rule A5-3-1**  
**Supports AUTOSAR19 Rule A5-3-1**
- 9415 'auto' variable *symbol* initialized using '*string*' list initialization**  
**note** Initializing an `auto` variable using list initialization can result in unexpected and compiler-dependent results. Additionally, the rules governing the type deduced when using list initialization with `auto` variables changed in C++17 which can result in code that has different meaning depending on the target language of the compiler. For example:
- ```

auto j1{1};           // direct list initialization
                      // j1 is std::initializer_list<int> in C++11 and int in C++17

auto j2 = {1};        // copy list initialization
                      // j2 is always std::initializer_list<int>
```

```

    auto j3 = 1;    // copy initialization - NOT list initialization
                   // j3 is always int

```

This message will be issued for the initialization of `j1` and `j2`. Avoiding list initialization for `auto` variables can prevent unintended results. The string parameter is either 'direct' or 'copy' indicating the type of list initialization employed (copy list initialization uses an equal sign, direct list initialization does not).

**Supports AUTOSAR17 Rule A8-5-3**

**Supports AUTOSAR19 Rule A8-5-3**

**9416 typedef used to define name *symbol***

**note** A typedef was used to define a type alias instead of a using alias.

**Supports AUTOSAR17 Rule A7-1-6**

**Supports AUTOSAR19 Rule A7-1-6**

**9417 data member *symbol* has protected access level**

**note** The specified *data* member of a class has an access level of `protected`. Some authors suggest against using `protected` data members.

**9418 enum *symbol* does not have an explicitly specified underlying type**

**note** Since C++11 it is possible to explicitly specify an underlying type for `enums` and enum classes. This message is issued for `enums` that do not explicitly specify an underlying type in their definition. This message is not issued in C or C++03 modes.

**Supports AUTOSAR17 Rule A7-2-2**

**Supports AUTOSAR19 Rule A7-2-2**

**9419 enum *symbol* is not a scoped enumeration**

**note** Scoped enumerations (those using `enum class` or `enum struct`) were introduced in C++11 and limit the scope of the enumeration constants to the enumeration. This message is issued for unscoped `enums`. This message is not issued in C or C++03 modes.

**Supports AUTOSAR17 Rule A7-2-3**

**Supports AUTOSAR19 Rule A7-2-3**

**9420 bitfield *symbol* does not have unsigned integer or explicitly unsigned enumeration type**

**note** This message is issued when a bitfield is declared with a type that is not an implementation-independent unsigned integral type. In particular, the message is issued for bitfields with plain `char`, plain `int`, or `wchar_t` types or types that are implicitly or explicitly signed such as `short` and `signed short`. For bitfields of enumeration type, if the underlying type of the enumeration is explicitly specified as an unsigned type, the message is suppressed. As it is only possible to specify an explicit underlying type for enumerations since C++11, the message will be issued for all bitfields of `enum` type in C or C++03 modes.

**Supports AUTOSAR17 Rule A9-6-1**

**9421 virtual function *symbol* overrides function *symbol* and is not marked with 'override' or 'final'**

**note** A virtual function that overrides a base class function was not declared with at least one of `override` or `final`. This message is only emitted for C++11 and higher. See also [1915](#) which is issued when an overriding function is not specified with `override`, even if `final` is specified.

**Supports AUTOSAR17 Rule A10-3-2**

**Supports AUTOSAR19 Rule A10-3-2**

**9422 virtual function *symbol* should specify exactly one of 'virtual', 'override', or 'final'**

**note** A virtual function was declared without specifying exactly one of `virtual`, `override`, or `final`. Some authors suggest that `virtual` should be specified in the base class and that overriding functions should be specified with either `override` or `final` and without a redundant `virtual` specifier. This message is only issued for C++11 and higher.

**Supports AUTOSAR17 Rule A10-3-1**

**Supports AUTOSAR19 Rule A10-3-1**

**9423 non-basic character used in identifier *string***

**note** This message is issued when a non-basic source character is used in an identifier.

**9424 parameter list omitted from lambda expression**

**note** The parentheses denoting an empty parameter list were absent from a lambda expression. C++ allows a lambda expression to be written without an explicit parameter list if the lambda takes no arguments, does not specify a trailing return type or an exception specification, and is not specified as `constexpr` or `mutable`. Including an empty parameter list provides visual consistency with other callable objects.

**Supports AUTOSAR17 Rule A5-1-3**

**Supports AUTOSAR19 Rule A5-1-3**

**9426 lambda used as operand to `decltype` or `typeid`**

**note** An expression of lambda type was used as an operand to the `decltype` or `typeid` operators. For example:

```
void f() {
    auto lambda = []() -> int { return 2; };
    using lambda_t = decltype(lambda);          // note 9426
}
```

Since each lambda has a distinct type, this usage is suspicious, for example, the `typeid` of two identically defined lambdas will be different.

**Supports AUTOSAR17 Rule A5-1-7**

**Supports AUTOSAR19 Rule A5-1-7**

**9428 object declared with the definition of *class/struct/union/enum symbol***

**note** This message is issued whenever an enumeration, union, class, or structure object is declared with the definition of its type. Some coding guidelines prohibit the definition of a class, structure, union, or enumeration type with the declaration of a variable in the same statement. For example:

```
enum class E1 { C11, C12, } e1;    // 9428
enum class E2 { C21, C22, };      // Okay
E2 e2;
struct S *s;                      // Okay - Not a definition of S
```

**Supports AUTOSAR19 Rule A7-1-9**

**9432 class *symbol* has multiple non-interface bases**

**note** An *interface* is a class where 1) all non-static member functions (if any) are public, virtual, and pure, and 2) all data members (if any) are public, static (or `thread_local`), and `constexpr` (or `const` in C++ modes prior to C++11). This message is issued when a class inherits from more than one non-interface class. The list of non-interface bases are provided in a supplemental message.

Supports AUTOSAR17 Rule A10-1-1

Supports AUTOSAR19 Rule A10-1-1

**9433 literal operator function *symbol* declared**

**note** This message is issued whenever a literal operator function declaration is encountered. Literal operators introduce user-defined literal suffixes that can be applied to certain types of literals. See also message [9434](#) which is issued when a literal operator is invoked by a user-defined literal.

Supports AUTOSAR17 Rule A13-1-1

**9434 user-defined literal with suffix *string* used**

**note** This message is issued when a user-defined literal is encountered. This message is parameterized by the suffix used and can be suppressed for individual suffixes using `-estring`. See also message [9433](#) which is issued for the declaration of literal operator functions which introduce user-defined suffixes.

Supports AUTOSAR17 Rule A13-1-1

**9435 *string symbol* declared as friend in class *symbol***

**note** This message is issued when a friend declaration is encountered. The *string* parameter is either `symbol` or `type` to indicate the kind of entity declared as a friend. The first *symbol* parameter is the entity declared as a friend unless the entity is a type in which case the parameter is actually a *type* parameter. The last *symbol* is the class in which the friend declaration appeared. Some authors suggest that friend declarations reduce encapsulation and should be avoided.

Supports AUTOSAR17 Rule A11-3-1

Supports AUTOSAR19 Rule A11-3-1

**9436 symbol *symbol* has array type *type***

**note** A symbol was declared with array type or reference to array type in a C++ module. Because arrays are implicitly convertible to pointers and don't maintain size information across such conversions, arrays are the subject of various programming errors. Some authors suggest replacing arrays with other container types such as `std::array` or `std::vector`. This message is not issued for `static constexpr` class data members (or `static const` data members in C++03 where `constexpr` is not available).

Supports AUTOSAR17 Rule A18-1-1

Supports AUTOSAR19 Rule A18-1-1

**9437 non-POD class *symbol* defined with 'struct' keyword**

**note** This message is issued when a non-POD class is defined using the `struct` keyword. Some authors suggest that non-POD types should be defined using the `class` keyword which forces private default access control.

Supports AUTOSAR17 Rule A11-0-1

Supports AUTOSAR19 Rule A11-0-1

**9438 non-copy, non-move assignment operator *symbol* should not be virtual**

**note** A non-copy, non-move assignment operator was declared as virtual; this is rarely the right thing to do. Virtual copy assignment operators are reported by [9407](#). Virtual move assignment operators are reported by [9410](#).

Supports AUTOSAR17 Rule A10-3-5

Supports AUTOSAR19 Rule A10-3-5

**9439 hexadecimal *integral/floating* literal '*string*' should be upper case**

**note** This message is issued whenever a hexadecimal floating or integer literal with lower case digits (a-f) is

encountered.

Supports AUTOSAR19 Rule A2-13-5

**9440 digit sequence separators in *string* *binary/octal/decimal/hexadecimal integer/floating* literal**  
**note** '*string*' should be every *integer* digits

This message is issued whenever the digit sequence separators (') in an integral or floating literal do not occur every Nth digit, where N is: 2 for hexadecimal, 3 for decimal and octal, and 4 for binary.

Supports AUTOSAR17 Rule A13-6-1

Supports AUTOSAR19 Rule A13-6-1

**9441 non-final class *symbol* has a public non-virtual destructor**

**note** This message is issued whenever there is a public non-virtual destructor in an essentially non-final class. A class is essentially non-final if neither it nor its destructor is marked **final**.

Supports AUTOSAR17 Rule A12-4-2

Supports AUTOSAR19 Rule A12-4-2

**9442 nested lambda expression**

**note** This message is issued whenever a lambda expression is nested in another lambda expression. For example:

```
void f() {
    auto f1 = []() {          // okay
        auto f2 = []() {};    // 9442
    };
}
```

Supports AUTOSAR17 Rule A5-1-8

Supports AUTOSAR19 Rule A5-1-8

**9443 universal-character-name used in identifier *string***

**note** This message is issued when a Universal Character Name (UCN) appears in an identifier. A UCN starts with \u followed by 4 hex digits or \U followed by 8 hex digits. Using UCNs in identifiers can make the identifiers difficult to read and differentiate from other identifiers.

Supports AUTOSAR19 Rule A2-13-6

**9444 user declared comparison operator *symbol* is a member function**

**note** This message is issued whenever a comparison operator (==, !=, <, <=, >=, >) is a member function. Some coding guidelines require comparison operators to be symmetric with respect to their operands. For example:

```
struct S {
    bool operator==(const S& o) const noexcept;           // 9444
    friend bool operator!=(const S& l, const S& r) noexcept; // Okay
};
```

Supports AUTOSAR19 Rule A13-5-5

**9445 user declared comparison operator *symbol* is not noexcept**

**note** This message is issued whenever a comparison operator (==, !=, <, <=, >=, >) is not **noexcept**. Some coding guidelines require comparison operators to be **noexcept**. For example:

```
struct S {
    friend bool operator==(const S& l, const S& r);           // 9445
};
```



```
    friend bool operator!=(const S& l, const S& r) noexcept;    // Okay
};
```

Supports AUTOSAR19 Rule A13-5-5

**9446 user declared comparison operator *symbol* does not have identical parameter types**

**note** This message is issued whenever a comparison operator (`==`, `!=`, `<`, `<=`, `>=`, `>`) has different parameter types. Some coding guidelines require comparison operators to be symmetric with respect to their operands. For example:

```
struct S {
    friend bool operator==(const S& l, const S* r) noexcept;    // 9446
    friend bool operator!=(const S& l, const S& r) noexcept;    // Okay
};
```

Supports AUTOSAR19 Rule A13-5-5

**9447 the return type of *non-member/member* assignment operator *symbol* should be *type***

**info** This message is issued whenever the return type of a user defined assignment operator is not:

- a reference to the class for member functions;
- a reference to the first parameter for non-member functions.

For example:

```
class C1 {
public:
    // ...
    C1& operator+=(const C1& rhs) &;    // okay
    C1 operator-=(const C1& rhs) &;    // 9447
};

class C2 { /* ... */ };

C2& operator+=(C2& lhs, const C2& rhs);    // okay
C2 operator-=(C2& lhs, const C2& rhs);    // 9447
```

Supports AUTOSAR17 Rule A13-2-1

Supports AUTOSAR19 Rule A13-2-1

**9448 the return value of *non-member/member* assignment operator *symbol* should be *string***

**info** This message is issued whenever the return value of a user defined assignment operator is not:

- `*this` for member functions;
- the first parameter for non-member functions.

For example:

```
class C3 {
public:
    // ...
    C3& operator+=(const C3& rhs) & {
        return *this;    // okay
    }

    C3& operator-=(C3& rhs) & {
```

```

        return rhs;                                // 9448
    }
};

class C4 { /* ... */ };

C4& operator+=(C4& lhs, const C4& rhs) {
    return lhs;                                    // okay
}

C4& operator-=(C4& lhs, C4& rhs) {
    return rhs;                                    // 9448
}

```

Supports AUTOSAR17 Rule A13-2-1

Supports AUTOSAR19 Rule A13-2-1

#### 9449 **inline** member function *symbol* is not marked as **inline**

**note** This message is issued whenever a non-deleted, non-defaulted, inline member function definition is not marked with the **inline**, **constexpr**, or **constexpr** keywords and it is not a function template or a member function of a class template. For example:

```

class A {
public:
    void f1() {}                                // 9449
    inline void f2() {}                        // okay - marked inline
    A() = default;                            // okay - marked default
    ~A() = delete;                            // okay - marked delete
    constexpr int f3() { return 2; }          // okay - marked constexpr
    consteval int f4() { return 3; }          // okay - marked consteval
    template <typename T>
    void f5(const T& t) {}                    // okay - function template
};

template <typename T>
class B {
public:
    explicit B(const T& x) : t(x) {}          // okay - member of class template
private:
    T t;
};

```

Supports AUTOSAR19 Rule A3-1-5

#### 9456 **do** statement used outside of statement-like macro

**note** This message reports any use of a **do** statement with the exception that a **do** statement with a condition of 0 or **false** may be used in the definition of a macro.

Supports AUTOSAR19 Rule A6-5-3

#### 9501 preprocessing directive in call to function *symbol*

**note** A preprocessing directive appeared within the call to a function, for example:

```

void init_buffer(void *buffer) {

```

```

    memset(buffer, 0,
#ifdef LARGE_BUFFER
    1024
#else
    128
#endif
    );
}

```

The problem is that if `memset` is implemented as a macro, the presence of the embedded preprocessor directives would invoke undefined behavior. PC-lint Plus issues message 436 when a preprocessor directive appears inside a macro invocation but message 9501 can be used to warn about directives in function calls that may be implemented as macros in other configurations or when compiled on other platforms.

One alternative is to move the directives outside the call, e.g.:

```

void init_buffer(void *buffer) {
    const unsigned buf_size =
#ifdef LARGE_BUFFER
    1024
#else
    128
#endif
    ;
    memset(buffer, 0, buf_size);
}

```

This message isn't issued if the function name is surrounded by parenthesis which suppresses the invocation of a potential macro by the same name. This message is parameterized on the function being called and can be enabled for specific functions using either `+esym` or `+ecall`.

**Supports CERT C PRE32-C** - *Do not use preprocessor directives in invocations of function-like macros*

#### 9502 multi-statement macro '*name*' is not enclosed in monocarpic do-while loop

**note**

A macro definition appeared to contain multiple statements but was not enclosed in a mono-carpic do/while loop. A macro is considered to contain multiple statements if the replacement text contains one or more semi-colons. A mono-carpic do/while loop is one that will be executed exactly once. PC-lint Plus is specifically expecting a loop of the following form:

```
#define M(x) do { ... } while (0)
```

where 0 may be replaced by any integer literal with a zero value (0u, 0x0, etc.), the `false` keyword or a keyword that has been assigned the same meaning as `false` with the `-rw_asgn` option, or an identifier whose name is `false`, ignoring case.

A macro that expands to multiple statements may result in unintentional interpretation during expansion. For example:

```
#define ADJUST(a, b) a = (b); a++; b = 0
```

would not work as intended if used as:

```

void foo(int a, int b) {
    if (a >= b)
        ADJUST(a, b);
}

```

which would expand to:

```
void foo(int a, int b) {
```

```

    if (a >= b)
        a = (b); a++; b = 0;
}

```

where the last two statements would be unconditionally executed as they are not part of the `if` statement. Simply wrapping the macro definition in braces is not always sufficient:

```
#define ADJUST(a, b) { a = (b); a++; b = 0; }
```

as this will prevent the macro from being usable in certain contexts, e.g.:

```

void foo(int a, int b) {
    if (a >= b)
        ADJUST(a, b);
    else
        ADJUST(b, a);
}

```

will result in a compile error as the `else` is not attached to the `if` statement because of the semi-colon. Wrapping the replacement text in a `do/while (0)` loop addresses all of the related issues:

```
#define ADJUST(a, b) do { a = (b); a++; b = 0; } while (0)
```

Supports CERT C PRE10-C - Wrap multistatement macros in a do-while loop

**9503** *note* declaration of *symbol* specifies an alignment of *integer* differently from another declaration that derives an equivalent alignment on a different basis (*string* vs *string*)

The resulting alignment of an externally visible object is consistent with other declarations of the object, but is not derived the same way. If the configuration or target architecture changes, there is a risk of introducing an inconsistency, resulting in undefined behavior. For example, if the size of `float` is 4:

```

extern _Alignas(4)    int a; // Note 9503
extern _Alignas(float) int a; // Note 9503

```

The related message [2502](#), will report when an externally visible object has a resulting alignment that conflicts with other declarations.

Supports MISRA C 2012 AMD3 Rule 8.15

**9504** *note* argument with essential type '*string*' supplied to type-generic macro '*macro*' resulting in call to function *symbol* should have an *string* type

The operand argument passed to a type-generic macro has an inappropriate *essential type*. The operand argument must have an essentially signed, essentially unsigned, or essentially (real or complex) floating type. Type-generic macros from `tgmath.h` can support real types, complex types, or both. Passing a complex type to a macro expecting a real type results in undefined behavior. Likewise, arguments of non-arithmetic types are not convertible to any of the corresponding real types defined for the macros and attempting to use them is also undefined behavior. This message overlaps with [2504](#). In addition to the undefined behavior that message reports, this includes violations of the MISRA C 2012 essential type system.

```

float i;
char c;
sqrt(f); // OK, essentially floating real type
exp(c);  // Note 9504, essentially character type

```

Supports MISRA C 2012 AMD3 Rule 21.22

**9901** *note* return value '*string*' for call to function *symbol* updated to '*string*' via return semantic '*string*'

This message is emitted when a return semantic adds or updates value tracking information for a return

value of a function call either because the semantic contains more specific information than was gleaned from walking the body of the called function or because the `fso` flag was set.

**9902** **return value '*string*' for call to function *symbol* not updated by return semantic '*string*' which adds no new information**

**note**

This message is emitted when a return semantic is not applied to a function call because the semantic does not provide any information that was not already known from walking the call.

**9903** ***essential-type-preview***

**note**

This message shows the step-by-step evaluation of how an expression's MISRA C 2012 *essential type* is calculated. See the `f12` flag for details.

**9904** **hook event: '*string*'**

**note**

This message is emitted every time a hookable event is reached in the AST walking phase.

**9905** **value tracking debug assertion not known to be unequivocally true**

**note**

(*Debug*) When an explicit cast is seen to a type defined by a `typedef` named `__vt_assert`, conventionally a `typedef` for `void`, this message will be issued during Value Tracking if it does not appear to be guaranteed that the subexpression is a non-zero integral value. This may occur multiple times for the same expression during specific walks.

**9906** **value tracking debug value: *string***

**note**

(*Debug*) When an explicit cast is seen to a type defined by a `typedef` named `__vt_alert`, conventionally a `typedef` for `void`, this message will be issued during Value Tracking parameterized by the value of the subexpression as it would be displayed in a Value Tracking message. This may occur multiple times for the same expression during specific walks.

**9907** **metric report debug value**

**note**

(*Debug*) If the report format `debug_msg` is specified using the `format` sub-option of the `+metric_report` option then message 9907 will be emitted with the text of each individual row that would have been emitted with `format=csv`.

## 23 Revision History

For the latest information on PC-lint Plus, please refer to [the news page on our website](#).

### 23.1 Version 2.2

#### 23.1.1 Highlights

PC-lint Plus 2.2 improves support for C++20, introduces support for C++23, addresses compatibility issues with recent versions of GCC and Visual Studio, and resolves several defects.

#### 23.1.2 Summary

##### 23.1.2.1 New Features

PCLP-4697     [Support for C++20 Module Partitions](#)

##### 23.1.2.2 Improvements

PCLP-3964     [Added support for C++20 literal class non-type template parameters to Value Tracking](#)  
 PCLP-4334     [Resolved syntax errors when using recently updated versions of Visual Studio 2022 standard library headers in C++20 mode](#)  
 PCLP-4647     [Improved instructions for integrating PC-lint Plus with the SEGGER IDE](#)  
 PCLP-4671     [Add support for C17, C23, and C++23 language standards](#)  
 PCLP-4708     [Improved type printing](#)  
 PCLP-4725     [Added detail parameters to message 540](#)  
 PCLP-4746     [Improved Strong Types in deduced types and template argument deduction](#)  
 PCLP-4750     [Improve filenames for `file` metrics](#)

##### 23.1.2.3 Bugs Fixed

PCLP-1471     [Resolved crash when missing enumeration constant is used in constexpr functions](#)  
 PCLP-1643     [Resolved crash when parsing overloaded shift operator with variadic arguments](#)  
 PCLP-4270     [Internal error D8BD27EC during instantiation of templated lambdas](#)  
 PCLP-4285     [Moved coroutine implementation out of experimental namespace](#)  
 PCLP-4331     [Resolved error when using a templated consteval function](#)  
 PCLP-4682     [Updated messages to support C++20 parenthesized list initialization](#)  
 PCLP-4699     [Fixed rendering of Unicode characters when dumping queries on Windows](#)  
 PCLP-4704     [Resolved false negative 578 for static local variables](#)  
 PCLP-4711     [Correct support for Keil `armcc` compiler](#)  
 PCLP-4726     [Fixed location information of message 631](#)  
 PCLP-4727     [Resolved false negative 9035 for VLA through typedef](#)  
 PCLP-4728     [Resolved false negative 971 when used as enum-base](#)  
 PCLP-4729     [Resolved false negative 9141 and 753 for friend class forward declarations](#)  
 PCLP-4730     [Resolved false negative 1901 and 1946 for variable templates](#)  
 PCLP-4731     [Resolved false negative 768 for multiple anonymous structs](#)  
 PCLP-4732     [Resolved false positive 9447 in template classes members with explicit template arguments](#)  
 PCLP-4733     [Resolved false positive 9447 with typedef return type](#)  
 PCLP-4734     [Resolved false positive 957 with internal linkage parameters](#)  
 PCLP-4735     [Resolved false positive 912 and 915 for enum with base promotable to int](#)  
 PCLP-4736     [Resolved false positive 911 for default template argument](#)  
 PCLP-4737     [Correct typename in messages 641 and 911 for enumerations with promotable integer base type](#)  
 PCLP-4738     [Change message number 541 to 2005](#)

**23.1.2.4 Known Issues**

|           |                                                                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-939  | Message 756 is disabled                                                                                                                       |
| PCLP-960  | Message 9103 not issued for inter-module reuse                                                                                                |
| PCLP-2135 | False negative 9003 for static variables                                                                                                      |
| PCLP-2603 | Message 767 is disabled                                                                                                                       |
| PCLP-2614 | False positive instance of message 522 for call to const member function but has external side-effects                                        |
| PCLP-3004 | Message 1565 does not report the status of sub-objects within data members of class type                                                      |
| PCLP-3198 | False positive 9045 for nested structures                                                                                                     |
| PCLP-3213 | False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter |
| PCLP-3223 | The undocumented <code>+fil</code> flag is not implemented                                                                                    |
| PCLP-3803 | False positive 650 for shifted values                                                                                                         |
| PCLP-3856 | False positive side-effect related messages for a lambda that has side effects                                                                |
| PCLP-4010 | False positive 9049 within lambda body                                                                                                        |
| PCLP-4046 | False positive 1764 for universal reference parameter of template instantiated with a lambda                                                  |

**23.1.3 New Features**


---

|           |                                                                                                                                |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|
| PCLP-4697 | <b>Support for C++20 Module Partitions</b><br>Support for C++20 Modules has been updated to add support for module partitions. |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|

**23.1.4 Improvements**


---

|           |                                                                                                                                                                                                                                                                                                                                                         |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3964 | <b>Added support for C++20 literal class non-type template parameters to Value Tracking</b><br>Value Tracking can now analyze values derived from non-type template arguments of literal class type, which were introduced in C++20.                                                                                                                    |
| PCLP-4334 | <b>Resolved syntax errors when using recently updated versions of Visual Studio 2022 standard library headers in C++20 mode</b><br>Updated support for C++20 has resolved syntax errors in Visual Studio 2022 library headers when used with <code>-std=c++20</code> including <code>&lt;type_traits&gt;</code> and headers specific to C++20 features. |
| PCLP-4647 | <b>Improved instructions for integrating PC-lint Plus with the SEGGER IDE</b><br>Updated the XML file used for integrating PC-lint Plus with the SEGGER IDE to correctly categorize messages.                                                                                                                                                           |
| PCLP-4671 | <b>Add support for C17, C23, and C++23 language standards</b><br>Support has been added for C17, C23, and C++23 language modes.                                                                                                                                                                                                                         |

- PCLP-4708     **Improved type printing**
- The tag type keyword will only be present when printing types if it is specified in the source code for struct, enum, union, and class in C++ mode.
  - Namespace qualification won't be added to types unless it is present in the source code.
  - Template class non-type default arguments are not printed unless they are specified to be a non-default value.
  - For an array type declared with `typedef`, the underlying type will also be printed after the typedef name.
  - Anonymous enums, structs, classes, and unions either printed as their symbol name if they were defined in a `typedef`, or as anonymous as well as the location of it's definition to differentiate different anonymous types.
  - The type of `nullptr` (`nullptr_t`) will be prefixed with "std::" in C++ mode.
  - Array type qualifiers now always come before the non-qualified element type, even if they are added implicitly or through `typedef`.
- PCLP-4725     **Added detail parameters to message 540**  
 Message 540 (initializer-string for char array is too long) now has an extra parameters to explain the difference in the array size and the initializer size.
- PCLP-4746     **Improved Strong Types in deduced types and template argument deduction**  
 Previously in the context of deduced (`auto`) types or template argument deduction, Strong Types (see section 7 Strong Types) would not be preserved. Now Strong Types will be inferred for deduced types.
- PCLP-4750     **Improve filenames for file metrics**  
 Filenames for file metrics no longer include current directory dot-slash prefixes and now respect the `+ffn` flag option.

### 23.1.5 Bugs Fixed

- 
- PCLP-1471     **Resolved crash when missing enumeration constant is used in constexpr functions**  
 PCLP-4723  
 Previously if an enumeration constant that was not defined, perhaps due to a misconfiguration, in the source code was used in a `constexpr` function, it would sometimes cause a crash. This is no longer the case.
- PCLP-1643     **Resolved crash when parsing overloaded shift operator with variadic arguments**  
 Previously parsing the definition of an overloaded shift operator with variadic arguments would cause a crash, but now it does not.
- PCLP-4270     **Internal error D8BD27EC during instantiation of templated lambdas**  
 Instantiation of templated lambdas could previously result in an internal error D8BD27EC. This issue has been corrected.



- PCLP-4285 **Moved coroutine implementation out of experimental namespace**  
Previously the C++20 standard header file `coroutine` declared its contents in the `std::experimental` namespace. Now it exists in the proper `std` namespace.
- PCLP-4331 **Resolved error when using a templated consteval function**  
Previously trying to evaluate a templated `constexpr` function in a template context would issue an error message. Now these evaluations will no longer issue an error message.
- PCLP-4682 **Updated messages to support C++20 parenthesized list initialization**  
Messages [414](#) (possible division by zero), [453](#) (impure function marked pure), [564](#) (result depends on order of evaluation), [1762](#) (member function could be const), and [9084](#) (assignment used in a larger expression) now account for parenthesized list initialization that C++20 introduced.
- PCLP-4699 **Fixed rendering of Unicode characters when dumping queries on Windows**  
When using `+dump_queries(unicode)`, previously some Unicode characters representing the tree structure of the query would be emitted as `?`. Now these characters are correctly emitted as UTF-8 encoded characters and can be viewed in terminals and text editors that support it.
- PCLP-4704 **Resolved false negative 578 for static local variables**  
Previously message [578](#) would not be emitted for static local variables, now a message will be emitted for them.
- PCLP-4711 **Correct support for Keil armcc compiler**  
The changes previously introduced in PCLP-4346 incorrectly targeted the Keil uVision `armclang` compiler instead of the `armcc` compiler. The compiler database entry for `keil_armcc` has been corrected.
- PCLP-4726 **Fixed location information of message 631**  
Message [631](#) (tagged symbol defined differently) previously would emit the location of a forward declaration in another module, not the definition. Now it will provide the location of the definition in another module.
- PCLP-4727 **Resolved false negative 9035 for VLA through typedef**  
Message [9035](#) (variable length array declared) will now be emitted for variable length arrays declared using a type defined with a `typedef`.
- PCLP-4728 **Resolved false negative 971 when used as enum-base**  
Message [971](#) (use of plain char) will now be emitted for enumerations that specify a plain `char` as their *enum-base*.
- PCLP-4729 **Resolved false negative 9141 and 753 for friend class forward declarations**  
Message [9141](#) (global declaration of symbol) and message [753](#) (local type not referenced) will now be emitted type declared using a friend class forward declaration.
- PCLP-4730 **Resolved false negative 1901 and 1946 for variable templates**  
Message [1901](#) (creating a temporary) and message [1946](#) (use of functional-style cast) will now be emitted in the context of variable templates.
- PCLP-4731 **Resolved false negative 768 for multiple anonymous structs**  
Message [768](#) (global structure member not referenced) would previously only emit for non-referenced members of the first encountered global anonymous struct. Now the message is emitted for all non-referenced members of all global anonymous structs.

- PCLP-4732     **Resolved false positive 9447 in template classes members with explicit template arguments**  
 Message 9447 (non-standard user defined assignment operator return) would previously be emitted in the context of template class assignment operator which specified a return type of that class explicitly using template arguments. In this case, the message is no longer emitted.
- PCLP-4733     **Resolved false positive 9447 with typedef return type**  
 Message 9447 (non-standard user defined assignment operator return) would previously be emitted when returning the class type defined with a `typedef`. In this case, the message is no longer emitted.
- PCLP-4734     **Resolved false positive 957 with internal linkage parameters**  
 Message 957 (function defined without a prototype) would previously be emitted for functions that have parameter types with internal linkage. In this case, the message is no longer emitted since those parameter types prevent the function from having external linkage.
- PCLP-4735     **Resolved false positive 912 and 915 for enum with base promotable to int**  
 Message 912 (arithmetic conversions applied to binary operand beyond integer promotion) and message 915 (implicit arithmetic conversion) will no longer be emitted for enumeration types with an *enum-base* type that would receive an integer promotion to the `int` type.
- PCLP-4736     **Resolved false positive 911 for default template argument**  
 Message 911 (implicit promotion) will no longer be emitted for default non-type template arguments in which the type is larger than the type of the expression provided.
- PCLP-4737     **Correct typename in messages 641 and 911 for enumerations with promotable integer base type**  
 Message 911 (implicit promotion) will now emit the enumeration's type instead of an underlying *enum-base* type. Message 641 (implicit conversion of enum to integral type) will now emit the underlying *enum-base* type after integer promotion in the context of a bitshift, if it is a type promotable to `int`, instead of just the underlying *enum-base* type.
- PCLP-4738     **Change message number 541 to 2005**  
 Previously when a character escape sequence was out of range warning 541 would be emitted, but now error 2005 is emitted. This was changed because the desired value of the character or string literal cannot be determined or inferred in this scenario.

### 23.1.6 Known Issues

- 
- PCLP-939     **Message 756 is disabled**  
 Message 756 (global typedef not referenced) is disabled in the current release.
- PCLP-960     **Message 9103 not issued for inter-module reuse**  
 Message 9103 (identifier with static storage reused) is not issued when a reuse of a static storage identifier occurs in a separate module.

- PCLP-2135     **False negative 9003 for static variables**  
 A file-scope static variable referenced only within a single function will not be reported by 9003. The message will be issued for variables that are implicitly or explicitly extern.
- PCLP-2603     **Message 767 is disabled**  
 Message 767 is disabled in the current release.
- PCLP-2614     **False positive instance of message 522 for call to const member function but has external side-effects**  
 A false positive instance of message 522 may be emitted for a call to a const member function despite the presence of external side-effects.
- PCLP-3004     **Message 1565 does not report the status of sub-objects within data members of class type**  
 Message 1565 will report when an initializer function has not initialized a non-static data member of the class of which the initializer function is a non-static member function. It will not report based on the status of non-static data members of other objects nested recursively as sub-objects of a non-static data member of class type.
- PCLP-3198     **False positive 9045 for nested structures**  
 Message 9045 (complete definition of symbol is unnecessary in this translation unit) may be incorrectly issued for structures whose only use appears within another structure.
- PCLP-3213     **False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter**  
 When providing a cast expression as a template argument to a non-type template parameter some messages described as reporting implicit conversions may be issued as if the cast operand were implicitly converted to the cast type.
- PCLP-3223     **The undocumented +fil flag is not implemented**  
 The +fil flag, which would control whether or not indentation checking is applied to labels, is not implemented. This unimplemented flag is not documented in the list of flag options, but a reference to it is made in section 13.4.
- PCLP-3803     **False positive 650 for shifted values**  
 Message 650 (constant out of range for operator) is sometimes incorrectly issued when the result of a shift expression is compared with a constant value.
- PCLP-3856     **False positive side-effect related messages for a lambda that has side effects**
- PCLP-4010     **False positive 9049 within lambda body**  
 Message 9049 (increment/decrement operator used in expression with other side-effects) is incorrectly issued for increment and decrement expressions appearing as part of a statement without additional side-effects within the body of a lambda expression.
- PCLP-4046     **False positive 1764 for universal reference parameter of template instantiated with a lambda**  
 Message 1764 (reference parameter of function could be reference to const) may be incorrectly issued when a function template is instantiated with a lambda as the argument corresponding to a universal reference parameter.

## 23.2 Version 2.1

### 23.2.1 Highlights

PC-lint Plus 2.1 includes several new features and numerous improvements including:

- Support for CWE weakness detection using `au-cwe.lnt`.
- Support for MISRA C 2012 AMD-3 and introductory support for AMD-4.
- Improvements to Metrics including the new `num_switch_cases`, `num_incoming_calls`, `num_calling_functions`, and `num_called_functions` function metrics.
- Improvements to Queries including support for non-persistent variables in query echoes, new type-related functions, and the ability to suppress Query execution in library regions.
- Introductory support for C++20 Modules.
- Support for generating compiler and project configurations for Green Hills compilers using `pclp_config`.
- Integration support for GHS MULTI IDE and Visual Studio Code.
- Improved support for using PC-lint Plus in conjunction with anti-virus tools.
- Optional shell-like parsing of compiler options with `pclp_config` which supports quoted arguments containing spaces.
- Improved support for recent versions of GCC.
- `pclp_config` support for SEGGER compilers and IDE.

### 23.2.2 Summary

#### 23.2.2.1 New Features

|           |                                                                                                                                                                             |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3086 | Add compiler configuration support for Green Hills compilers                                                                                                                |
| PCLP-3607 | Add compiler configuration support for SEGGER compilers                                                                                                                     |
| PCLP-3763 | Introductory support for CWE                                                                                                                                                |
| PCLP-3874 | Support for C++20 Modules <code>module</code> and <code>import</code> syntax and new <code>-build_module_interface_unit</code> option to build C++20 module interface units |
| PCLP-4039 | New function metrics for incoming and outgoing function calls                                                                                                               |
| PCLP-4256 | Support C++20 <code>using enum</code>                                                                                                                                       |
| PCLP-4352 | New <code>fql</code> flag option to control Query execution in library regions                                                                                              |
| PCLP-4382 | New metric <code>function.num_switch_cases</code>                                                                                                                           |
| PCLP-4396 | Support MISRA C 2012 AMD-4 Rule 9.6 using messages 2903 and 2904                                                                                                            |

#### 23.2.2.2 Improvements

|           |                                                                                                                                                                    |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-1519 | Add Boolean function result inferencing for Value Tracking                                                                                                         |
| PCLP-2197 | Add anchored expressions to Value Tracking messages to enable <code>-equery</code> suppressions                                                                    |
| PCLP-3197 | False negative instance of message 9034 involving compound assignment                                                                                              |
| PCLP-3344 | Add <code>pclp_config</code> version to generated compiler configuration header files                                                                              |
| PCLP-3351 | Improve side effect classification for GNU Statement Expressions                                                                                                   |
| PCLP-3622 | Document integration with Visual Studio Code                                                                                                                       |
| PCLP-3623 | Fix typo in custom message text for BARR-C 5.1a                                                                                                                    |
| PCLP-3689 | Correct omission of documentation for existing C++98 support in description of <code>-std</code> and fix various trivial documentation typos                       |
| PCLP-3780 | Expand recognition of Boolean type hierarchies defined using Strong Types when determining whether an expression is Boolean for the purposes of MISRA C 2004 rules |

|           |                                                                                                                                                            |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3811 | Add <code>no_specific_walk</code> semantic, a new name for the previously undocumented <code>nowalk</code> semantic                                        |
| PCLP-3840 | Add support for <code>asm</code> and <code>__asm</code> keywords for IAR compiler configs                                                                  |
| PCLP-3859 | Resolve additional spurious pre-run crash messages caused by remote code injection by antivirus software                                                   |
| PCLP-3873 | Improve compiler configuration support for Microchip XC8                                                                                                   |
| PCLP-3888 | Added missing thread analysis semantic documentation                                                                                                       |
| PCLP-3903 | Improved <code>pclp_config</code> support for extracting compiler predefined macros                                                                        |
| PCLP-4002 | Support <code>__atomic_is_lock_free</code> built-in for sizes up to long long                                                                              |
| PCLP-4231 | Updated descriptions for messages 9225 and 1502                                                                                                            |
| PCLP-4235 | Expanded description of <code>+fnr</code> to clarify that it specifically applies to functions not available to be walked                                  |
| PCLP-4238 | New <code>--shell-parse-compiler-options</code> option for <code>pclp_config.py</code>                                                                     |
| PCLP-4245 | Add instructions for locating GCC compiler for STM32CubeIDE users                                                                                          |
| PCLP-4248 | Recognize <code>TI -include_path</code> options with unquoted paths                                                                                        |
| PCLP-4250 | Updated descriptions of message 759 and the <code>flo</code> flag option                                                                                   |
| PCLP-4251 | Expanded support for <code>__int128</code> , <code>__int128_t</code> , and <code>__uint128_t</code>                                                        |
| PCLP-4262 | Add version number, date, and revision history to "Using PC-lint Plus in Safety Critical Applications" document                                            |
| PCLP-4266 | New <code>faa</code> flag option controls aligned allocations                                                                                              |
| PCLP-4272 | New Query functions for working with types                                                                                                                 |
| PCLP-4291 | Support accessing contextual values of non-persistent variables within the body of <code>echo</code> expressions in Queries                                |
| PCLP-4301 | Support for MISRA C 2012 AMD-3                                                                                                                             |
| PCLP-4302 | Add new message 9202 to support MISRA C 2012 AMD-3 Rule 6.3                                                                                                |
| PCLP-4303 | Add new message 9203 to support MISRA C 2012 AMD-3 Rule 8.16                                                                                               |
| PCLP-4304 | Add new message 9205 to support MISRA C 2012 AMD-3 Rule 8.17                                                                                               |
| PCLP-4305 | Add new message 841 to support MISRA C 2012 AMD-3 Rule 17.10                                                                                               |
| PCLP-4306 | Add new message 9211 to support MISRA C 2012 AMD-3 Rule 23.1                                                                                               |
| PCLP-4307 | Add new message 2419 and 9213 to support MISRA C 2012 AMD-3 Rule 23.2                                                                                      |
| PCLP-4308 | Add new message 9208 to support MISRA C 2012 AMD-3 Rule 23.3                                                                                               |
| PCLP-4309 | Add new message 2416 to support MISRA C 2012 AMD-3 Rule 23.4                                                                                               |
| PCLP-4310 | Add new message 9210 to support MISRA C 2012 AMD-3 Rule 23.8                                                                                               |
| PCLP-4312 | Add new message 2505 to support MISRA C 2012 AMD-3 Rule 7.5                                                                                                |
| PCLP-4313 | Add new messages 2502 and 9503 reporting alignment conflicts to support MISRA C 2012 AMD-3 Rule 8.15                                                       |
| PCLP-4314 | Add new message 2417 to support MISRA C 2012 AMD-3 Rule 17.13                                                                                              |
| PCLP-4316 | Add new messages 2504 and 9504 to support MISRA C 2012 AMD-3 Rule 21.22                                                                                    |
| PCLP-4317 | Add new message 9218 to support MISRA C 2012 AMD-3 Rule 21.23                                                                                              |
| PCLP-4318 | Add new message 9214 to support MISRA C 2012 AMD-3 Rule 23.5                                                                                               |
| PCLP-4319 | Add new message 9216 to support MISRA C 2012 AMD-3 Rule 23.6                                                                                               |
| PCLP-4320 | Add new message 9219 to support MISRA C 2012 AMD-3 Rule 23.7                                                                                               |
| PCLP-4321 | Updates to <code>au-misra3.lnt</code> and <code>au-misra3-amd2.lnt</code> as per MISRA C 2012 Amendment 3                                                  |
| PCLP-4322 | Add complex floating essential type for MISRA C 2012 AMD-3                                                                                                 |
| PCLP-4326 | Add support for Visual Studio <code>/FI</code> and <code>/external:I</code> options                                                                        |
| PCLP-4330 | Correct false positive 779 messages issued for type-dependent expressions                                                                                  |
| PCLP-4337 | Detect character size when generating compiler configurations for TI compilers                                                                             |
| PCLP-4340 | Exclude <code>ti_edg</code> implementation detail compiler database entry from the list of compilers shown by <code>pclp_config</code>                     |
| PCLP-4342 | Add support for compiler relative include file search directories.                                                                                         |
| PCLP-4343 | Remove temporary compiler generated files within the configuration utility <code>pclp_config</code>                                                        |
| PCLP-4344 | Improved and corrected determining of compiler versions within the <code>pclp_config</code> utility.                                                       |
| PCLP-4349 | Improved error messages for <code>pclp_config</code> when a command uses <code>--compiler-database</code> where <code>--compilation-db</code> was intended |
| PCLP-4353 | Updated manufacturer of the Freescale compilers                                                                                                            |
| PCLP-4355 | Correct minor issues in generated Visual Studio compiler configuration files                                                                               |

|           |                                                                                                                           |
|-----------|---------------------------------------------------------------------------------------------------------------------------|
| PCLP-4358 | Improved Visual Studio configuration file comments                                                                        |
| PCLP-4387 | Include right shifts for message 9053                                                                                     |
| PCLP-4389 | Message 527 now reports code that is unreachable due to a prior infinite loop                                             |
| PCLP-4390 | Recognize the assignment of a constant value in a Boolean context as itself representing a constant value                 |
| PCLP-4392 | Support MISRA C 2012 AMD-4                                                                                                |
| PCLP-4398 | Support MISRA C 2012 AMD 4 Rule 18.10                                                                                     |
| PCLP-4409 | Support MISRA C 2012 AMD 4 Rule 12.6                                                                                      |
| PCLP-4416 | Improved <code>pclp_config</code> support for Visual Studio compiler options that start with a dash                       |
| PCLP-4425 | Revise when thread analysis processing is inhibited                                                                       |
| PCLP-4428 | New message 1797 improves support for MISRA C++ Rule 14-5-3 and AUTOSAR M14-5-3                                           |
| PCLP-4429 | Add warning message for when an object is used as a mutex and then as a locker.                                           |
| PCLP-4436 | Enable <code>include_next</code> preprocessor keyword in TI compiler configurations generated by <code>pclp_config</code> |
| PCLP-4453 | Examples added to description of the <code>-cond</code> option                                                            |
| PCLP-4459 | Note the constraints of <code>-setenv</code> when used in a configuration file                                            |
| PCLP-4491 | Recognize C++23 feature test macros in <code>pclp_config</code>                                                           |
| PCLP-4538 | Additional instructions for integrating PC-lint Plus with IAR Embedded Workbench                                          |
| PCLP-4590 | Improved link navigation for Revision History entries                                                                     |
| PCLP-4595 | Improved instructions for integrating PC-lint Plus with Visual Studio Code                                                |
| PCLP-4606 | Updates to <code>au-misra3.lnt</code> and <code>au-misra3-amd2.lnt</code> as per MISRA C 2012 Amendment 4                 |
| PCLP-4660 | Improve Keil ARMCC and Microchip XC8 compiler configuration generation                                                    |
| PCLP-4669 | Improved support for Microchip XC8 keywords and predefined macros                                                         |

### 23.2.2.3 Bugs Fixed

|           |                                                                                                                                                                                                                      |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3201 | Correct false negative 9102 for digraph sequence "<:"                                                                                                                                                                |
| PCLP-3551 | False positive 725 within template instantiations                                                                                                                                                                    |
| PCLP-3569 | Correct false positive 550 for braced member-init-lists                                                                                                                                                              |
| PCLP-3660 | Correct false positive 776 for the summation of two ints cast to a long of the same size                                                                                                                             |
| PCLP-4230 | False positive message 9006 for dependent <code>sizeof</code> expressions                                                                                                                                            |
| PCLP-4243 | Correct rare memory corruption in thread analysis semantic processing                                                                                                                                                |
| PCLP-4244 | Correct rare memory corruption in file stream semantic processing                                                                                                                                                    |
| PCLP-4246 | Resolve crash/internal error 292B4782 for constant division by zero                                                                                                                                                  |
| PCLP-4254 | Resolve false positive instances of messages 550, 551, and 552 for rewritten comparison operators in C++20                                                                                                           |
| PCLP-4255 | Correct handing of C/C++ options when determining size options in <code>pclp_config</code>                                                                                                                           |
| PCLP-4258 | Update <code>pclp_config</code> to prefer setting <code>size_t</code> to <code>unsigned long</code> over <code>unsigned long long</code> when both types are the same size to match recommendation from C++ standard |
| PCLP-4265 | False negative messages 1524, 1732, and 1733 for new expression appearing in a member init list                                                                                                                      |
| PCLP-4269 | Memory leak when using <code>-cond</code> and <code>-egrep/+egrep</code> options                                                                                                                                     |
| PCLP-4273 | Correct false positive message 504 for shift assignment operators                                                                                                                                                    |
| PCLP-4275 | Resolve internal error 187F3BF3 (-37)                                                                                                                                                                                |
| PCLP-4297 | False negative 785 for uninitialized members of a structure nested within a union                                                                                                                                    |
| PCLP-4300 | Correct name of the <code>file</code> argument of the <code>thread_report</code> options in built in help and the Reference Manual.                                                                                  |
| PCLP-4336 | Remove unimplemented <code>pclp_config</code> option <code>--ignore-options</code>                                                                                                                                   |
| PCLP-4341 | Resolve a crash/internal error/hang that could occur during Global Wrap-up when using the Metrics feature on a large project using many concurrent analysis threads                                                  |
| PCLP-4345 | Remove inapplicable C++ configuration from the Microchip XC8 and XC16 compiler database entries                                                                                                                      |
| PCLP-4346 | Correct handing of C/C++ options when determining size options in <code>pclp_config</code>                                                                                                                           |
| PCLP-4350 | Correct false positive 9107 for class template static data members                                                                                                                                                   |



|           |                                                                                                                                                                                                          |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-4351 | Resolve internal errors when evaluating certain Hook fields or Query functions for value-dependent expressions                                                                                           |
| PCLP-4361 | Correct false positive 9102 for digraph sequence "<::"                                                                                                                                                   |
| PCLP-4375 | Incorrect deprecation text for AUTOSAR Rule M18-7-1                                                                                                                                                      |
| PCLP-4414 | Updated <code>pclp_config</code> to handle fixed precision floating point types in standard header files                                                                                                 |
| PCLP-4446 | Internal error when the <code>isCXX11ConstantExpr</code> Query function is evaluated in a C module                                                                                                       |
| PCLP-4463 | Incorrect library region determination for certain Hook events                                                                                                                                           |
| PCLP-4464 | User-defined hooks executed twice for label statements                                                                                                                                                   |
| PCLP-4516 | Corrected invalid string escape sequences in <code>pclp_config</code>                                                                                                                                    |
| PCLP-4534 | Resolve Python error while generating include options when using <code>pclp_config.py</code> to generate a compiler configuration without specifying a compiler binary using <code>--compiler-bin</code> |
| PCLP-4567 | Resolved an issue in Value Tracking that manifested as a hang before crashing due to memory exhaustion                                                                                                   |
| PCLP-4577 | Inappropriate evaluation of Query echo reflections                                                                                                                                                       |
| PCLP-4597 | Resolved false negatives of message 550 and 551 due to suppressions                                                                                                                                      |
| PCLP-4630 | Enable usage of <code>_Float16</code> and <code>__bf16</code> types                                                                                                                                      |
| PCLP-4636 | Enable usage of <code>_Complex_Float16</code> type                                                                                                                                                       |
| PCLP-4679 | Resolve false negatives in the value of the <code>file.num_include_directives</code> metric in Linux and macOS builds of PC-lint Plus                                                                    |

#### 23.2.2.4 Known Issues

|           |                                                                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-939  | Message 756 is disabled                                                                                                                       |
| PCLP-960  | Message 9103 not issued for inter-module reuse                                                                                                |
| PCLP-2135 | False negative 9003 for static variables                                                                                                      |
| PCLP-2603 | Message 767 is disabled                                                                                                                       |
| PCLP-2614 | False positive instance of message 522 for call to const member function but has external side-effects                                        |
| PCLP-3004 | Message 1565 does not report the status of sub-objects within data members of class type                                                      |
| PCLP-3198 | False positive 9045 for nested structures                                                                                                     |
| PCLP-3213 | False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter |
| PCLP-3223 | The undocumented <code>+fil</code> flag is not implemented                                                                                    |
| PCLP-3803 | False positive 650 for shifted values                                                                                                         |
| PCLP-3856 | False positive side-effect related messages for a lambda that has side effects                                                                |
| PCLP-4010 | False positive 9049 within lambda body                                                                                                        |
| PCLP-4046 | False positive 1764 for universal reference parameter of template instantiated with a lambda                                                  |

#### 23.2.2.5 AUTOSAR Summary

|           |                                                                           |
|-----------|---------------------------------------------------------------------------|
| PCLP-4361 | Correct false positive 9102 for digraph sequence "<::"                    |
| PCLP-4230 | False positive message 9006 for dependent <code>sizeof</code> expressions |
| PCLP-3201 | Correct false negative 9102 for digraph sequence "<:"                     |

#### 23.2.2.6 MISRA Summary

|           |                                                                                 |
|-----------|---------------------------------------------------------------------------------|
| PCLP-4428 | New message 1797 improves support for MISRA C++ Rule 14-5-3 and AUTOSAR M14-5-3 |
| PCLP-4409 | Support MISRA C 2012 AMD 4 Rule 12.6                                            |
| PCLP-4398 | Support MISRA C 2012 AMD 4 Rule 18.10                                           |
| PCLP-4396 | Support MISRA C 2012 AMD-4 Rule 9.6 using messages 2903 and 2904                |

|           |                                                                                                                                                                    |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-4392 | Support MISRA C 2012 AMD-4                                                                                                                                         |
| PCLP-4387 | Include right shifts for message 9053                                                                                                                              |
| PCLP-4350 | Correct false positive 9107 for class template static data members                                                                                                 |
| PCLP-4322 | Add complex floating essential type for MISRA C 2012 AMD-3                                                                                                         |
| PCLP-4321 | Updates to <code>au-misra3.lnt</code> and <code>au-misra3-amd2.lnt</code> as per MISRA C 2012 Amendment 3                                                          |
| PCLP-4320 | Add new message 9219 to support MISRA C 2012 AMD-3 Rule 23.7                                                                                                       |
| PCLP-4319 | Add new message 9216 to support MISRA C 2012 AMD-3 Rule 23.6                                                                                                       |
| PCLP-4318 | Add new message 9214 to support MISRA C 2012 AMD-3 Rule 23.5                                                                                                       |
| PCLP-4317 | Add new message 9218 to support MISRA C 2012 AMD-3 Rule 21.23                                                                                                      |
| PCLP-4314 | Add new message 2417 to support MISRA C 2012 AMD-3 Rule 17.13                                                                                                      |
| PCLP-4313 | Add new messages 2502 and 9503 reporting alignment conflicts to support MISRA C 2012 AMD-3 Rule 8.15                                                               |
| PCLP-4312 | Add new message 2505 to support MISRA C 2012 AMD-3 Rule 7.5                                                                                                        |
| PCLP-4310 | Add new message 9210 to support MISRA C 2012 AMD-3 Rule 23.8                                                                                                       |
| PCLP-4309 | Add new message 2416 to support MISRA C 2012 AMD-3 Rule 23.4                                                                                                       |
| PCLP-4308 | Add new message 9208 to support MISRA C 2012 AMD-3 Rule 23.3                                                                                                       |
| PCLP-4307 | Add new message 2419 and 9213 to support MISRA C 2012 AMD-3 Rule 23.2                                                                                              |
| PCLP-4306 | Add new message 9211 to support MISRA C 2012 AMD-3 Rule 23.1                                                                                                       |
| PCLP-4305 | Add new message 841 to support MISRA C 2012 AMD-3 Rule 17.10                                                                                                       |
| PCLP-4304 | Add new message 9205 to support MISRA C 2012 AMD-3 Rule 8.17                                                                                                       |
| PCLP-4303 | Add new message 9203 to support MISRA C 2012 AMD-3 Rule 8.16                                                                                                       |
| PCLP-4302 | Add new message 9202 to support MISRA C 2012 AMD-3 Rule 6.3                                                                                                        |
| PCLP-4301 | Support for MISRA C 2012 AMD-3                                                                                                                                     |
| PCLP-4230 | False positive message 9006 for dependent <code>sizeof</code> expressions                                                                                          |
| PCLP-3780 | Expand recognition of Boolean type hierarchies defined using Strong Types when determining whether an expression is Boolean for the purposes of MISRA C 2004 rules |
| PCLP-3201 | Correct false negative 9102 for digraph sequence "<:"                                                                                                              |
| PCLP-3197 | False negative instance of message 9034 involving compound assignment                                                                                              |
| PCLP-4606 | Updates to <code>au-misra3.lnt</code> and <code>au-misra3-amd2.lnt</code> as per MISRA C 2012 Amendment 4                                                          |
| PCLP-4316 | Add new messages 2504 and 9504 to support MISRA C 2012 AMD-3 Rule 21.22                                                                                            |
| PCLP-4010 | False positive 9049 within lambda body                                                                                                                             |

### 23.2.3 New Features

---

|           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3086 | <b>Add compiler configuration support for Green Hills compilers</b><br>Compiler configurations for the Green Hill Software compiler drivers can now be generated with the <code>pclp_config</code> installation and configuration utility. All of the compiler drivers are supported via a <code>--compiler</code> option value of either <code>ghs</code> for mixed or C++ specific source files or <code>ghs_c</code> for C specific source files that uses any GHS C headers that have conflicts with the corresponding C++ headers. |
| PCLP-3607 | <b>Add compiler configuration support for SEGGER compilers</b><br>Compiler configurations for the SEGGER's <code>gcc</code> and <code>segger-cc</code> can now be generated using the <code>pclp_config.py</code> utility. Sections 2.3.11 and 2.3.20 were added to provide instructions on the process of generating a compiler configuration and integrating PC-lint Plus with the SEGGER IDE.                                                                                                                                        |
| PCLP-3763 | <b>Introductory support for CWE</b><br>The new <code>au-cwe.lnt</code> author file (in the <code>lnt/</code> directory of the PC-lint Plus distribution) enables support for CWE. In addition, a CWE Coverage Claim Representation ( <code>au-cwe.xml</code> ) is also available in the <code>lnt/</code> directory.                                                                                                                                                                                                                    |



- PCLP-3874 Support for C++20 Modules `module` and `import` syntax and new `-build_module_interface_unit` option to build C++20 module interface units**  
 The new option `-build_module_interface_unit(filename[, module_decl_name])` can now be used to build C++20 module interface units to be imported by `import` or `module` declarations in other translation units. The optional second argument must be used to provide the name of the module if the module name differs from the file's base name. Support for C++20 Modules in PC-lint Plus is currently limited to the subset of C++20 Modules functionality provided by clang 12. In particular, module partitions are not supported.
- PCLP-4039 New function metrics for incoming and outgoing function calls**  
 The new function metrics `num_incoming_calls`, `num_calling_functions`, and `num_called_functions` now provide access to the number of incoming calls, the number of distinct functions making at least one incoming call, and the number of distinct functions called by a particular function.
- PCLP-4256 Support C++20 using `enum`**  
 The C++20 using `enum` feature is now supported.
- PCLP-4352 New `fql` flag option to control Query execution in library regions**  
 The new `fql` flag option may be used to disable Queries evaluation in library regions.
- PCLP-4382 New metric function `num_switch_cases`**  
 The new metric function `num_switch_cases` represents the total number of `case` and `default` cases across all `switch` statements within a function.
- PCLP-4396 Support MISRA C 2012 AMD-4 Rule 9.6 using messages 2903 and 2904**  
 The new message 2903 reports when an initializer list mixes chained designators and positional initializers. The new message 2904 reports when a sub-object is initialized via braced initializer list with at least one positional init and a chained designator somewhere else in the containing init list. Together, these new messages implement MISRA C AMD-4 Rule 9.6.

#### 23.2.4 Improvements

- 
- PCLP-1519 Add Boolean function result inferencing for Value Tracking**  
**PCLP-4422** Support for inferencing in Value Tracking has been extended to improve support for  
**PCLP-4040** common smart pointer usage patterns with the recognition of conditional dereferences in the scope of pointer tests made indirectly using Boolean functions (including `operator bool`). Additionally, Value Tracking now walks rewritten binary operators generated by reliance on three-way comparison functions.
- PCLP-2197 Add anchored expressions to Value Tracking messages to enable `-equery` suppressions**  
 Value Tracking messages now provide anchored expressions when available to enable AST-based suppressions using `-equery`. Anchor information for an emitted message can be found using the `+paraminfo` option.

- PCLP-3197 False negative instance of message 9034 involving compound assignment**  
False negative instances of message [9034](#) for compound assignment operators assigning to objects with types ranked lower than `int` and `unsigned int` have been resolved. Previously, the messages affected by this false negative were only emitted when the option `-misra_interpret(c2012,essential type differs from standard type only for int and unsigned int)` was used.
- PCLP-3344 Add `pclp_config` version to generated compiler configuration header files**  
The `pclp_config` utility now includes a comment with version information in generated compiler configuration header files.
- PCLP-3351 Improve side effect classification for GNU Statement Expressions**  
The classification of top-level and inner side effects for the GNU Statement Expression language extension has been improved to resolve false positive instances of [523](#) for statement expressions with inner side effects. This also resolves false negative instances of message [522](#) for top-level statement expressions and related side effect interactions for other messages such as [9007](#).
- PCLP-3622 Document integration with Visual Studio Code**  
Visual Studio Code is now included among the list of IDEs with PC-lint Plus integration instructions in the Reference Manual.
- PCLP-3623 Fix typo in custom message text for BARR-C 5.1a**  
An extra space following the typedef name included within single quotes in the text of the custom message 8527 emitted by a `-astquery` option in `au-barr.lnt` has been corrected.
- PCLP-3689 Correct omission of documentation for existing C++98 support in description of `-std` and fix various trivial documentation typos**  
The description of the `-std` option has been updated to correct the previous omission of `c++98` from the documented list of valid arguments to this option. Several trivial spelling and grammar issues throughout the documentation have also been fixed.
- PCLP-3780 Expand recognition of Boolean type hierarchies defined using Strong Types when determining whether an expression is Boolean for the purposes of MISRA C 2004 rules**  
One of the methods by which an expression could be considered Boolean for the purposes of messages used to support MISRA C 2004 rules 12.6 ([9232](#)), 13.1 ([9236](#)), 13.2 ([9224](#)), and 15.4 ([9238](#)) is through the Strong Type of the expression. Previously, the Strong Type of the expression would only cause the expression to be considered Boolean for this purpose if the Strong Type of the expression was the designated Strong Boolean Type (set by a `-strong` option using either the `B` or `b` flag). This Boolean classification now also recognizes an expression whose Strong Type is not identical to the Strong Boolean Type but is convertible to it through a type hierarchy as Boolean for the purpose of these rules.
- PCLP-3811 Add `no_specific_walk` semantic, a new name for the previously undocumented `nowalk` semantic**  
Previous versions of PC-lint Plus recognized an undocumented function semantic `nowalk` which could inhibit Value Tracking from performing specific walks of a function while still permitting the general walk to occur. This semantic has been renamed `no_specific_walk` and documented. The previous name is retained as an alias for compatibility.

- PCLP-3840 Add support for `asm` and `__asm` keywords for IAR compiler configs**  
IAR compiler configurations did not previously include the `__asm` keyword as an option to specify in-line assembly. This change adds support for this keyword. Additionally, support for both `asm` and `__asm` was added to the IAR 8051 compiler configuration.
- PCLP-3859 Resolve additional spurious pre-run crash messages caused by remote code injection by antivirus software**  
**PCLP-4047**  
Spurious non-fatal crash messages with exception code 0x4001000A or 0x40010006 caused by remote code injection by certain antivirus software will no longer be emitted.
- PCLP-3873 Improve compiler configuration support for Microchip XC8**  
The `pclp_config` compiler database has been updated to improve support for the Microchip XC8 compiler when generating a compiler configuration.
- PCLP-3888 Added missing thread analysis semantic documentation**  
The thread analysis function semantics `async`, `thread_lock`, `thread_unlock`, `thread(thread_name)`, and `thread_mono(thread_name)` were not documented. The reference manual has been updated.
- PCLP-3903 Improved `pclp_config` support for extracting compiler predefined macros**  
The extracted predefined macros could be incomplete (Visual Studio) or absent (Microchip XC8). Newer versions of Visual Studio provide a method to dump predefined macros. This new method is used and if it fails the prior method is used. Instructions for the Microchip XC8 predefined macros has been added.
- PCLP-4002 Support `__atomic_is_lock_free` built-in for sizes up to long long**  
The `__atomic_is_lock_free` built-in used by standard library implementations now returns true for sizes up to the size of long long.
- PCLP-4231 Updated descriptions for messages 9225 and 1502**  
The description of message 9225 has been updated to correct a typo and the description of message 1502 updated to more accurately articulate the message parameterizations.
- PCLP-4235 Expanded description of `+fnr` to clarify that it specifically applies to functions not available to be walked**  
The description of the `fnr` flag now notes that it applies only to functions not available to be walked and describes when this occurs.
- PCLP-4238 New `--shell-parse-compiler-options` option for `pclp_config.py`**  
The `pclp_config.py` script now supports the new option `--shell-parse-compiler-options` which can be used to opt-in to shell-like parsing of the arguments for `--compiler-options`, `--compiler-c-options`, and `--compiler-cpp-options`. The behavior of this shell-like parsing is controlled by the `--posix-command-parsing` and `--no-posix-command-parsing` options.
- PCLP-4245 Add instructions for locating GCC compiler for STM32CubeIDE users**  
Section 2.3.3 now includes instructions for extracting the location of the GCC compiler from within the STM32CubeIDE development tool.
- PCLP-4248 Recognize TI `--include_path` options with unquoted paths**  
When generating compiler configurations for TI compilers, providing an unquoted path with the option `--include_path` resulted in a failure to represent that path in the generated `.lnt` file. This change resolves the issue, allowing unquoted file paths to be used with `--include_path`.

- PCLP-4250 Updated descriptions of message 759 and the `flo` flag option**  
Message 759 (header declaration for symbol could be moved from header to module) may be issued within a library region if the referenced symbol is a non-library symbol, such as occurs when a declaration for the symbol also appears in a non-library region when the `flo` flag option is OFF (the default). The description of message 759 previously implied that the message would never be issued within a library region. The description of the `flo` flag option now notes that library suppressions are not applied in certain situations.
- PCLP-4251 Expanded support for `__int128`, `__int128_t`, and `__uint128_t`**  
The non-standard `__int128` keyword and the `__int128_t` and `__uint128_t` typedefs are now supported by default. Support was previously only enabled when using `-sp8` or `-sp16`.
- PCLP-4262 Add version number, date, and revision history to "Using PC-lint Plus in Safety Critical Applications" document**  
The "Using PC-lint Plus in Safety Critical Applications" document (`doc/pclp-sca.pdf`) now includes a version number and date on the title page and concludes with a new revision history section.
- PCLP-4266 New `faa` flag option controls aligned allocations**  
A new flag `faa` has been added to enable or disable C++17 aligned allocations for a project.
- PCLP-4272 New Query functions for working with types**  
New Query functions have been added to provide direct access to fundamental types (`getVoidType`, `getBoolType`, `getCharType`, etc.), obtaining the size of types (`getTypeSizeInBits` and `getTypeSizeInChars`), converting between bits and char units (`toCharUnitsFromBits` and `toBitsFromCharUnits`), obtaining the canonical type of a given type (`getCanonicalType`), and obtaining the integral type corresponding to a specific bit width (`getSignedIntTypeForBitwidth` and `getUnsignedIntTypeForBitwidth`).
- PCLP-4291 Support accessing contextual values of non-persistent variables within the body of `echo` expressions in Queries**  
The values of non-persistent variables at the time that an `echo` was registered can now be accessed during the later evaluation of the `echo` body.
- PCLP-4301 Support for MISRA C 2012 AMD-3**  
MISRA C 2012 Amendment 3 is now supported using the provided `au-misra3-amd3.lnt` file found in the `lnt/` directory of the PC-lint Plus distribution.
- PCLP-4302 Add new message 9202 to support MISRA C 2012 AMD-3 Rule 6.3**  
The new message 9202 (bitfield declared as member of union) reports member bitfields declared in unions and supports MISRA C 2012 AMD-3 Rule 6.3.
- PCLP-4303 Add new message 9203 to support MISRA C 2012 AMD-3 Rule 8.16**  
The new message 9203 (declaration contains an alignment attribute whose expression evaluates to zero) reports on declarations containing alignment attributes with zero-valued expressions and supports MISRA C 2012 AMD-3 Rule 8.16.
- PCLP-4304 Add new message 9205 to support MISRA C 2012 AMD-3 Rule 8.17**  
The new message 9205 (declaration contains multiple alignment attributes) reports when a declaration is encountered containing multiple alignment attributes and supports MISRA C 2012 AMD-3 Rule 8.17.

- PCLP-4305     **Add new message 841 to support MISRA C 2012 AMD-3 Rule 17.10**  
The new message 841 (function is declared as noreturn but returns a value) reports when a function is declared as noreturn but whose return type is not void and supports MISRA C 2012 AMD-3 Rule 17.10.
- PCLP-4306     **Add new message 9211 to support MISRA C 2012 AMD-3 Rule 23.1**  
The new message 9211 (generic selection is not expanded from a function-like macro) reports when a generic selection is encountered that is not expanded from a macro or when the controlling expression is not listed as one of the macro's arguments. This implements MISRA C 2012 AMD-3 Rule 23.1.
- PCLP-4307     **Add new message 2419 and 9213 to support MISRA C 2012 AMD-3 Rule 23.2**  
The new message 2419 (controlling expression contains a side effect which will not be evaluated) reports when a controlling expression contains a side effect. The new message 9213 (controlling expression contains a call to function which will not be called) reports when a controlling expression contains a function call. This implements MISRA C 2012 AMD-3 Rule 23.2.
- PCLP-4308     **Add new message 9208 to support MISRA C 2012 AMD-3 Rule 23.3**  
The new message 9208 (generic selection does not contain any non-default associations) reports when a generic selection is encountered that only contains a default association and supports MISRA C 2012 AMD-3 Rule 23.3.
- PCLP-4309     **Add new message 2416 to support MISRA C 2012 AMD-3 Rule 23.4**  
The new message 2416 (association list contains inappropriate type) reports when a generic selection is encountered that contains an association which can never be selected. This implements MISRA C 2012 AMD-3 Rule 23.4.
- PCLP-4310     **Add new message 9210 to support MISRA C 2012 AMD-3 Rule 23.8**  
The new message 9210 (generic selection does not contain any non-default associations) reports when a generic selection is encountered that contains a default association that does not come first or last in the association list and supports MISRA C 2012 AMD-3 Rule 23.8.
- PCLP-4312     **Add new message 2505 to support MISRA C 2012 AMD-3 Rule 7.5**  
The new message 2505 (argument to integer constant macro should be an unsuffixed integer literal) reports when an integer constant macro uses an inappropriate type. This implements MISRA C 2012 AMD-3 Rule 7.5.
- PCLP-4313     **Add new messages 2502 and 9503 reporting alignment conflicts to support MISRA C 2012 AMD-3 Rule 8.15**  
The new messages 2502 (differing alignment requirements seen) and 9503 (declaration specifies an alignment differently from another declaration that derives an equivalent alignment on a different basis) report when an object is declared multiple times with different alignment requirements. These messages provide partial support for MISRA C 2012 AMD-3 Rule 8.15.
- PCLP-4314     **Add new message 2417 to support MISRA C 2012 AMD-3 Rule 17.13**  
The new message 2417 (const/volatile on function type has unspecified behavior) reports when a function type is specified with const or volatile. This new message, combined with existing errors 4175 and 5805, supports MISRA C 2012 AMD-3 Rule 17.13.

- PCLP-4316     **Add new messages 2504 and 9504 to support MISRA C 2012 AMD-3 Rule 21.22**  
 The new message 9504 (argument supplied to type-generic macro resulting in call to function should have an essentially signed, essentially unsigned, or essentially (real or complex) floating type) reports when a type-generic macro from `<tmath.h>` is encountered that does not use an appropriate essential type. This implements MISRA C 2012 AMD-3 Rule 21.22. 2504 (argument supplied to type-generic macro resulting in call to function should have a real or complex arithmetic type) is also introduced to warn about such uses that result in undefined behavior.
- PCLP-4317     **Add new message 9218 to support MISRA C 2012 AMD-3 Rule 21.23**  
 The new message 9218 (arguments to type-generic macro from `tmath.h` have differing types) reports when a multi-argument `tmath` macro call is encountered that uses varying types. This implements MISRA C 2012 AMD-3 Rule 21.23.
- PCLP-4318     **Add new message 9214 to support MISRA C 2012 AMD-3 Rule 23.5**  
 The new message 9214 (generic selection selects default because pointer type is not implicitly converted) reports when a generic selection is encountered that selects the default association when an implicit pointer conversion might have been expected. Such cases where a pointer association differs in qualifiers, implicit casts to void, or null constants selecting default are reported. This supports MISRA C 2012 AMD-3 Rule 23.5.
- PCLP-4319     **Add new message 9216 to support MISRA C 2012 AMD-3 Rule 23.6**  
 The new message 9216 (essential type of the controlling expression does not match its standard type) reports when a generic selection is encountered whose controlling expression has an essential type that differs from its standard type. This message provides support for MISRA C 2012 AMD-3 Rule 23.6.
- PCLP-4320     **Add new message 9219 to support MISRA C 2012 AMD-3 Rule 23.7**  
 The new message 9219 (parameter of generic selection macro is not evaluated exactly once) reports when a generic selection macro is encountered that does not evaluate the arguments in the controlling expression exactly once, regardless of which association is selected. This supports MISRA C 2012 AMD-3 Rule 23.7.
- PCLP-4321     **Updates to `au-misra3.lnt` and `au-misra3-amd2.lnt` as per MISRA C 2012 Amendment 3**  
 MISRA C 2012 Amendment 3 includes several updates to MISRA C 2012 and MISRA C 2012 AMD-2 that were not previously reflected in the corresponding `au-misra3.lnt` and `au-misra3-amd2.lnt` files. These updates include reclassifying Rule 21.11 as Advisory, reclassifying Rule 21.12 as Required, removal of the restrictions for the `_Generic` keyword and the `stdalign.h` and `stdnoreturn.h` headers for Rule 1.4, and increasing the scope of Rule 21.12 to include all facilities of the `fenv.h` header. These changes are now reflected in the `au-misra3.lnt` and `au-misra3-amd2.lnt` files.
- PCLP-4322     **Add complex floating essential type for MISRA C 2012 AMD-3**  
 MISRA C 2012 essential type messages now support a new "complex floating" essential type category to support MISRA C 2012 AMD-3.
- PCLP-4326     **Add support for Visual Studio `/FI` and `/external:I` options**  
 The Visual Studio `/FI` and `/external:I` compiler options are now supported.
- PCLP-4330     **Correct false positive 779 messages issued for type-dependent expressions**  
 Message 779 (string constant in comparison operator) was previously incorrectly issued for type-dependent expressions. This issue has been resolved.



- PCLP-4337 Detect character size when generating compiler configurations for TI compilers**  
Generated compiler configurations for TI compilers (`ti_cl2000`, `ti_cl430`, `ti_cl6x`, and `ti_armcl`) now extract the target architecture character size from the compiler and add an appropriate `-sb` option in the generated size options. This will not affect the previous behavior for most compilers which will simply generate a `-sb8` option, but it is particularly relevant to the `ti_cl2000` compiler entry where a `-sb16` option will now be generated to correctly support the TI cl2000 compiler targeting 16-bit characters.
- PCLP-4340 Exclude `ti_edg` implementation detail compiler database entry from the list of compilers shown by `pclp_config`**  
The `ti_edg` entry in the `pclp_config` compiler database is an abstract base and implementation detail for other, specific TI compilers and is not generally intended for direct use. This compiler database entry will no longer be included in the output from `--list-compilers`.
- PCLP-4342 Add support for compiler relative include file search directories.**  
The `keil_armcc`, `microchip_xc8`, and TI compilers search for include files in known directories relative to the compiler but do not have a method to extract the directories. The compiler configuration include file search options will now be determined from the location of the compiler.
- PCLP-4343 Remove temporary compiler generated files within the configuration utility `pclp_config`**  
When `pclp_config` invokes the user selected compiler, many compilers generate files associated with the provided source files. The invocation instructions in the compilers database did not always have the proper instructions to delete all of the compiler generated file. All compiler generated temporary files will now be deleted.
- PCLP-4344 Improved and corrected determining of compiler versions within the `pclp_config` utility.**  
The `pclp_config` utility was unable to determine the compiler version for some of the supported compilers. The missing instructions were added and the incorrect instructions were updated. The compiler version can now be determined for all `pclp_config` supported compilers .
- PCLP-4349 Improved error messages for `pclp_config` when a command uses `--compiler-database` where `--compilation-db` was intended**  
The `pclp_config.py` option `--compiler-database` is used to specify an alternative path to the `compilers.yaml` compiler database which provides information about the compilers supported by `pclp_config`. It is typically unnecessary to use this option to specify the path explicitly because the `compilers.yaml` compiler database included in the PC-lint Plus distribution is automatically loaded from the same directory as the `pclp_config.py` script if the directory structure has not been modified. The distinct and unrelated option `--compilation-db` is used when generating a project configuration to specify the path to a JSON compilation database such as a `compile_commands.json` compilation database generated by CMake.
- The error messages emitted in cases where `pclp_config.py` is invoked with an argument that appears to have specified one of these two options where the other was likely intended have been improved to specifically highlight this distinction.
- PCLP-4353 Updated manufacturer of the Freescale compilers**  
Updated the manufacture of the Freescale compilers to be NXP/Freescale.

- PCLP-4355 Correct minor issues in generated Visual Studio compiler configuration files**  
The generated Visual Studio compiler configuration files did not suppress messages 13 and 104 in `uchar.h` (Visual Studio 2015 and later) and `ucyvalshar.h` (Visual Studio 2013); did not suppress message 5881 in `setjmp.h` and `eh.h` (Visual Studio 2015); did not suppress message 5394 in `scoped_allocator` (Visual Studio 2012); did not suppress message 4886 in `math.h` (Visual Studio 2005); and did not enable support for C++20 for Visual Studio 2019. In addition, if the user supplied compiler options included the `/JMC` option, it was treated as a `J` option. All of these issues have been corrected. Additionally, `/Zc:wchar_t` and `/Zc:wchar_t-` C++ options are now correctly translated.
- PCLP-4358 Improved Visual Studio configuration file comments**  
The comments for specific Visual Studio options in the `pclp_config` generated compiler configuration files are now more readable and understandable.
- PCLP-4387 Include right shifts for message 9053**  
The message 9053, implemented to support MISRA C 2012 Rule 12.2, did not report when excessive right shifts occurred. Additionally, 9053 did not report negative shifts of any kind. Excessive right shift and negative shift operations will now be reported by message 9053.
- PCLP-4389 Message 527 now reports code that is unreachable due to a prior infinite loop**  
Message 527 now reports statements that are unreachable due to appearing after an infinite loop with a constant condition without a `break`.
- PCLP-4390 Recognize the assignment of a constant value in a Boolean context as itself representing a constant value**  
Message 506 now recognizes the result of the assignment of a constant as yielding a constant value. This message will now report constructs such as `while (x = 1)` where an assignment of a constant appears in a Boolean context.
- PCLP-4392 Support MISRA C 2012 AMD-4**  
MISRA C 2012 Amendment 4 is now supported using the provided `au-misra3-amd4.lnt` file found in the `lnt/` directory of the PC-lint Plus distribution.
- PCLP-4398 Support MISRA C 2012 AMD 4 Rule 18.10**  
The new message 2502 (pointer to variably modified array type was used) reports when a pointer whose pointee is a variably modified array type is declared. This implements MISRA C AMD-4 Rule 18.10
- PCLP-4409 Support MISRA C 2012 AMD 4 Rule 12.6**  
The new message 181 (atomic record type directly accessed) reports when a member of an atomic struct or union is accessed via the `.` or `->` operators. This implements MISRA C AMD-4 Rule 12.6.
- PCLP-4416 Improved `pclp_config` support for Visual Studio compiler options that start with a dash**  
Visual Studio compiler options may start with either a forward slash (`/`) or a dash (`-`). Previously `pclp_config` only recognized the `-I` compiler option and ignored any other compiler options that started with a dash. Recognized Visual Studio compiler options may now start with either a forward slash or a dash.



- PCLP-4425 **Revise when thread analysis processing is inhibited**  
The second phase of thread analysis, which occurs after Global Wrap-up, previously would not execute if there were fewer than 2 threads (or a single non-mono thread). This caused the thread reports and beneficial thread analysis messages to not be emitted. Assuming that Global Wrapup is not disabled, the thread reports will now always be generated and the thread analysis second phase will be inhibited only if there are no threads or mutexes, or if a thread analysis fatal condition has occurred unless overridden by setting the `ftc` flag to 2 or greater.
- PCLP-4428 **New message 1797 improves support for MISRA C++ Rule 14-5-3 and**  
PCLP-4417 **AUTOSAR M14-5-3**  
The new message [1797](#) replaces the previous use of message [1721](#) as the support mechanism for MISRA C++ Rule 14-5-3 and AUTOSAR M14-5-3. The previous use of message 1721 to support this rule resulted in false positive rule violation messages because message 1721 reports assignment operators that are not copy nor move assignment operators while the rule only applies if the assignment operator is further a function template taking a parameter whose type is a template parameter type. Message 1797 reports the specific situation described in the rule. The level of support for AUTOSAR M14-5-3 has been changed from "Partially supported" to "Supported".
- PCLP-4429 **Add warning message for when an object is used as a mutex and then as a locker.**  
When an object is initially used as a mutex and subsequently used as a locker, the usages as a locker were silently ignored. This condition will now be reported with message [2540](#) and will no longer inhibit the second phase of thread analysis.
- PCLP-4436 **Enable `include_next` preprocessor keyword in TI compiler configurations generated by `pclp_config`**  
Recent versions of the TI compilers support for the `include_next` preprocessor directive and use it in their C++ headers to include their corresponding C headers. `pclp_config` now enables support for the `include_next` preprocessor for the TI compilers.
- PCLP-4453 **Examples added to description of the `-cond` option**  
Examples have been added to the description of the `-cond` option.
- PCLP-4459 **Note the constraints of `-setenv` when used in a configuration file**  
Environment variable expansions are not affected by `-setenv` options appearing within the same configuration file. This is now noted in the section [4.1.11 Expansion of Environment Variables in Options](#).
- PCLP-4491 **Recognize C++23 feature test macros in `pclp_config`**  
All C++23 feature test macros are recognized by `pclp_config` when generating the compiler configuration header file.
- PCLP-4538 **Additional instructions for integrating PC-lint Plus with IAR Embedded Workbench**  
Section [2.3.15](#) is changed to include additional setup instructions that show how to Run PC-lint Plus using a keyboard shortcut. The additional instructions also highlight how to add PC-lint Plus to the "Toolbar".
- PCLP-4590 **Improved link navigation for Revision History entries**  
The clickable links in the summary sections of the Revision History now navigate directly to the corresponding issue instead of the beginning of the containing section.

- PCLP-4595 **Improved instructions for integrating PC-lint Plus with Visual Studio Code**  
Section 2.3.19 has been reworked to include additional setup instructions, add support for supplemental messages, overcome "Problem Matcher" limitations, and more.
- PCLP-4606 **Updates to au-misra3.lnt and au-misra3-amd2.lnt as per MISRA C 2012 Amendment 4**  
MISRA C 2012 Amendment 4 includes several updates to MISRA C 2012 and MISRA C 2012 AMD-2 that were not previously reflected in the corresponding au-misra3.lnt and au-misra3-amd2.lnt files. These updates include an exception to Rule 3.1 to allow uniform resource identifiers within comments, and removing the prohibition of <stdatomic.h> and <threads.h> library facilities from Rule 1.4. These changes are now reflected in the au-misra3.lnt and au-misra3-amd2.lnt files.
- PCLP-4660 **Improve Keil ARMCC and Microchip XC8 compiler configuration generation**  
The configuration utility (pclp\_config) did not include all of the compiler supplied header directories in the generated compiler configurations for the Keil ARMCC and Microchip XC8 compilers. The compilers.yaml file includes instruction comments if the assumptions for the required headers for the Keil ARMCC and Microchip XC8 compilers are incorrect and need to be adjusted by the user.
- PCLP-4669 **Improved support for Microchip XC8 keywords and predefined macros**  
Errors were previously emitted when analyzing several of the Microchip XC8 header files because of misconfigured keywords and predefined macros. The generated XC8 compiler configurations have been corrected to eliminate these issues.

### 23.2.5 Bugs Fixed

- 
- PCLP-3201 **Correct false negative 9102 for digraph sequence "<:"**  
The message 9102 (possible digraph sequence) was not correctly emitted when encountering the character sequence <: and digraphs were enabled.
- PCLP-3551 **False positive 725 within template instantiations**  
Message 725 (unexpected lack of indentation) would sometimes incorrectly be issued for the body of a *constexpr if* statement within an instantiated template. Messages 525 (unexpected negative indentation) and 725 are no longer emitted for template instantiations.
- PCLP-3569 **Correct false positive 550 for braced member-init-lists**  
The message 550 (local variable not subsequently accessed) was incorrectly emitted when encountering an initializer list using braced syntax.
- PCLP-3660 **Correct false positive 776 for the summation of two ints cast to a long of the same size**  
The message 776 (possible truncation of addition) was unjustly emitted when long and int are the same size.
- PCLP-4230 **False positive message 9006 for dependent sizeof expressions**  
Message 9006 (sizeof used on expression with side effect) was previously incorrectly emitted for value-dependent sizeof expressions. This issue has been corrected.

- PCLP-4243 **Correct rare memory corruption in thread analysis semantic processing**  
Combinations of several thread analysis semantics and certain rare code sequences could cause memory corruption and incorrect processing of the semantics of the affected user function. The memory corruption has been corrected which also corrects the processing of the semantics.
- PCLP-4244 **Correct rare memory corruption in file stream semantic processing**  
Combinations of several file stream semantics and certain rare code sequences could cause memory corruption and incorrect processing of the semantics of the affected user function. The memory corruption has been corrected which also corrects the processing of the semantics.
- PCLP-4246 **Resolve crash/internal error 292B4782 for constant division by zero**  
A crash (Windows Exception Code 0xC0000094) or an internal error (292B4782) which could occur when a constant expression involving guaranteed division (or remainder) by zero (e.g. `1 / 0`) appeared in certain contexts has been resolved.
- PCLP-4254 **Resolve false positive instances of messages 550, 551, and 552 for rewritten comparison operators in C++20**  
Previously in C++20 mode, if a variable was only accessed using a comparison operator in the *rewritten candidate set* (a rewritten comparison operator), a false positive 550, 551, or 552 could later be emitted for the variable. This behavior has been updated to recognize variable accesses within operands to rewritten comparison operators.
- PCLP-4255 **Correct handling of C/C++ options when determining size options in pclp\_config**  
The `pclp_config` configuration utility could incorrectly handle the user provided compiler options while extracting the information needed for the size options.
- PCLP-4258 **Update pclp\_config to prefer setting `size_t` to unsigned long over unsigned long long when both types are the same size to match recommendation from C++ standard**  
The C++ standard recommend that `size_t` should be a type whose integer conversion ranks is no greater than that of `signed long int` unless a larger size is necessary to represent all possible values. Prior versions of the compiler database used by `pclp_config.py` would preferentially configure the size type to `unsigned long long` instead of `unsigned long` when `sizeof(long) == sizeof(long long)`. This could cause false positive 4589 (enum redeclaration with different underlying type) messages to be emitted after encountering a conflicting definition of `size_t` in a standard library header. The compiler database has been updated to prefer `unsigned long` in this situation.
- PCLP-4265 **False negative messages 1524, 1732, and 1733 for new expression appearing in a member init list**  
PC-lint Plus previously did not emit messages 1524 (new in constructor for class which has no explicit destructor), 1732 (new in constructor for class which has no user-provided copy assignment operator), and 1733 (new in constructor for class which has no user-provided copy constructor) for `new` expressions appearing in a constructor's member init list. This has been corrected.
- PCLP-4269 **Memory leak when using `-cond` and `-egrep/+egrep` options**  
A memory leak which could occur when using the `-cond` and `-egrep/+egrep` options has been corrected.

- PCLP-4273 **Correct false positive message 504 for shift assignment operators**  
Message 504 (unusual shift operation) was previously incorrectly issued for shift assignment operators with unparenthesized right-hand operands. This issue has been resolved.
- PCLP-4275 **Resolve internal error 187F3BF3 (-37)**  
An internal error with error code 187F3BF3 and diagnostic info -37 that could occur during the evaluation of a non-static data member initializer containing an implicit or explicit use of the `this` pointer outside the context of a non-static member function has been resolved.
- PCLP-4297 **False negative 785 for uninitialized members of a structure nested within a union**  
Message 785 (too few initializers) was not previously emitted for initializer lists of aggregates nested within a union. This has been corrected.
- PCLP-4300 **Correct name of the file argument of the thread\_report options in built in help and the Reference Manual.**  
Some instances of the `file` argument of the `thread_report` options were incorrectly referred to as `filename`.
- PCLP-4336 **Remove unimplemented pclp\_config option --ignore-options**  
The `pclp_config` option `--ignore-options` was recognized but never implemented. The option has been removed.
- PCLP-4341 **Resolve a crash/internal error/hang that could occur during Global Wrap-up when using the Metrics feature on a large project using many concurrent analysis threads**  
An issue that could variously manifest as a crash, a hang, or an internal error during Global Wrap-up when analyzing a large project using both the Metrics feature and enabling many concurrent threads using the `-max_threads` option has been resolved.
- PCLP-4345 **Remove inapplicable C++ configuration from the Microchip XC8 and XC16 compiler database entries**  
The Microchip XC8 and XC16 compilers only support C code. Unnecessary configuration entries related to extracting C++ information from these compilers have been removed from `compilers.yaml`.
- PCLP-4346 **Correct handing of C/C++ options when determining size options in pcp\_config**  
The `pcp_config` configuration utility was unable to extract the information to generate the size options for all compilers.
- PCLP-4350 **Correct false positive 9107 for class template static data members**  
Message 9107 (header cannot be included in more than one translation unit) was incorrectly emitted when multiple definitions of a class template static data member were encountered.

- PCLP-4351 Resolve internal errors when evaluating certain Hook fields or Query functions for value-dependent expressions**  
 Use of the `is_constant_expr`, `is_ice`, or `is_value` Hook fields or the `isConstantExpr`, `isIntegerConstantExpr`, `evaluateAsBooleanCondition`, `evaluateAsFloat`, `evaluateAsInt`, `getIntegerConstantExpr`, `evaluateAsBooleanCondition`, `isCXX11ConstantExpr`, or `isCXX98IntegralConstantExpr` Query functions would previously elicit an internal error when evaluated for a value-dependent expression. This issue has been corrected.
- PCLP-4361 Correct false positive 9102 for digraph sequence "<::"**  
 The message 9102 (possible digraph sequence) was incorrectly emitted when encountering the character sequence `<::` not immediately followed by `:` or `>`.
- PCLP-4375 Incorrect deprecation text for AUTOSAR Rule M18-7-1**  
 The `au-autosar.lnt` and `au-autosar19.lnt` author files previously incorrectly referred to AUTOSAR Rule M18-7-1 as A18-7-1 in the deprecation text issued with message 586. The deprecation text used in these files has been corrected.
- PCLP-4414 Updated `pclp_config` to handle fixed precision floating point types in standard header files**  
 Previously when projects using C++20 in GCC 12 or later, configurations generated by `pclp_config` would cause a parse issue due to the use of the `__float80` type in the `<compare>` header file. Additionally when projects using C++ code in GCC 13 or later combined with glibc 2.37 or later, configurations generated by `pclp_config` would cause a parse issues due to the use of the `_Float32`, `_Float32x`, `_Float64`, `_Float64x`, and `_Float128` types in various glibc header files. New compiler configurations generated by `pclp_config` will not emit these parse errors.
- PCLP-4446 Internal error when the `isCXX11ConstantExpr` Query function is evaluated in a C module**  
 Evaluation of the `isCXX11ConstantExpr` Query function for an expression appearing in a C module would previously result in internal error 1AF10F50. This issue has been resolved.
- PCLP-4463 Incorrect library region determination for certain Hook events**  
 Previously, user-defined hooks executed for statement events would sometimes utilize the incorrect library status resulting in incorrect results for expansion of the `is_library` field as well as messages emitted from such hooks being inappropriately suppressed.
- PCLP-4464 User-defined hooks executed twice for label statements**  
 Previously, user-defined hooks would be executed twice for label statements. This issue has been corrected.
- PCLP-4516 Corrected invalid string escape sequences in `pclp_config`**  
 The automated configuration tool `pclp_config` contained invalid string escape sequences that prior to version 3.12 of python were silently ignored. These invalid string escape sequences were corrected.

- PCLP-4534     **Resolve Python error while generating include options when using pclp\_config.py to generate a compiler configuration without specifying a compiler binary using --compiler-bin**  
 The error message `Error: local variable 'found_paths' referenced before assignment` derived from a Python exception within `pclp_config.py` previously prevented the generation of an output file when running `pclp_config.py` without specifying a compiler binary with `--compiler-bin`. Attempting to generate a compiler configuration without specifying a compiler binary is intended to produce several warning messages noting the functionality that is unavailable without the compiler binary, but the process now correctly produces a limited output file without an error.
- PCLP-4567     **Resolved an issue in Value Tracking that manifested as a hang before crashing due to memory exhaustion**  
 An issue that could cause processing to hang before crashing once memory was exhausted during Value Tracking has been resolved.
- PCLP-4577     **Inappropriate evaluation of Query echo reflections**  
 Previously, an `echo` expression's reflection would sometimes incorrectly be evaluated in the context of a subsequently defined query which could result in unexpected behavior or a crash. This has been corrected.
- PCLP-4597     **Resolved false negatives of message 550 and 551 due to suppressions**  
 For message 550 (local variable not accessed) a false negative could previously occur if message 550 was suppressed before the end of a function scope. For message 551 (static non-local variable not accessed) a false negative could previously occur if message 551 was suppressed before the end of the file. Both have been resolved.
- PCLP-4630     **Enable usage of \_Float16 and \_\_bf16 types**  
 Previous use of the `_Float16` or `__bf16` types would result in a parse error. Now no error is produced when using these types.
- PCLP-4636     **Enable usage of \_Complex\_Float16 type**  
 Previous use of the `_Complex_Float16` type would result in a parse error. Now no error is produced when using this type.
- PCLP-4679     **Resolve false negatives in the value of the file.num\_include\_directives metric in Linux and macOS builds of PC-lint Plus**  
 An issue that previously could have resulted in the `file.num_include_directives` metric undercounting include directives when using the Linux or macOS builds of PC-lint Plus has been resolved. This issue did not impact Windows builds of PC-lint Plus.

### 23.2.6 Known Issues

- 
- PCLP-939     **Message 756 is disabled**  
 Message 756 (global typedef not referenced) is disabled in the current release.
- PCLP-960     **Message 9103 not issued for inter-module reuse**  
 Message 9103 (identifier with static storage reused) is not issued when a reuse of a static storage identifier occurs in a separate module.

- PCLP-2135     **False negative 9003 for static variables**  
 A file-scope static variable referenced only within a single function will not be reported by 9003. The message will be issued for variables that are implicitly or explicitly extern.
- PCLP-2603     **Message 767 is disabled**  
 Message 767 is disabled in the current release.
- PCLP-2614     **False positive instance of message 522 for call to const member function but has external side-effects**  
 A false positive instance of message 522 may be emitted for a call to a const member function despite the presence of external side-effects.
- PCLP-3004     **Message 1565 does not report the status of sub-objects within data members of class type**  
 Message 1565 will report when an initializer function has not initialized a non-static data member of the class of which the initializer function is a non-static member function. It will not report based on the status of non-static data members of other objects nested recursively as sub-objects of a non-static data member of class type.
- PCLP-3198     **False positive 9045 for nested structures**  
 Message 9045 (complete definition of symbol is unnecessary in this translation unit) may be incorrectly issued for structures whose only use appears within another structure.
- PCLP-3213     **False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter**  
 When providing a cast expression as a template argument to a non-type template parameter some messages described as reporting implicit conversions may be issued as if the cast operand were implicitly converted to the cast type.
- PCLP-3223     **The undocumented +fil flag is not implemented**  
 The +fil flag, which would control whether or not indentation checking is applied to labels, is not implemented. This unimplemented flag is not documented in the list of flag options, but a reference to it is made in section 13.4.
- PCLP-3803     **False positive 650 for shifted values**  
 Message 650 (constant out of range for operator) is sometimes incorrectly issued when the result of a shift expression is compared with a constant value.
- PCLP-3856     **False positive side-effect related messages for a lambda that has side effects**
- PCLP-4010     **False positive 9049 within lambda body**  
 Message 9049 (increment/decrement operator used in expression with other side-effects) is incorrectly issued for increment and decrement expressions appearing as part of a statement without additional side-effects within the body of a lambda expression.
- PCLP-4046     **False positive 1764 for universal reference parameter of template instantiated with a lambda**  
 Message 1764 (reference parameter of function could be reference to const) may be incorrectly issued when a function template is instantiated with a lambda as the argument corresponding to a universal reference parameter.



## 23.3 Version 2.0

### 23.3.1 Highlights

PC-lint Plus 2.0 introduces major new features:

- **Metrics** provides an intuitive and flexible solution for verifying, reporting, and extending over one hundred code metrics including cyclomatic complexity, number of paths, categorized line and statement counts, and Halstead measures. Metrics are available at the project, module, file, class, and function level. PC-lint Plus Metrics make it easy to check for violations of metric thresholds or constraints, create custom metrics, and calculate derived information such as minimum, maximum, and average metric values. Metric reports are available in XML, JSON, and CSV format. For more details, see the [Metrics](#) chapter of the Reference Manual.
- **Queries** enables the creation of entirely new custom messages directly within PC-lint Plus configuration files using a convenient syntax operating on a parsed AST representation. Custom messages can be used to automatically check your own novel coding guidelines or add specialized analysis to validate the usage of project-specific APIs. Queries also enable an unprecedented level of control over message suppression by providing access to AST information which can examine, e.g., whether a member function is virtual or whether a class has any data members when determining whether a specific message should be emitted or suppressed. For more details, see the [PC-lint Plus Query Language](#) chapter of the Reference Manual.

This release of PC-lint Plus also includes:

- Improved support for the latest C++ standards
- Support for Visual Studio 2022
- Support for AUTOSAR19 and new AUTOSAR messages
- Refinements to MISRA support
- Performance improvements
- ...and many other improvements discussed in the detailed revision history below

### 23.3.2 Summary

#### 23.3.2.1 New Features

|           |                                                          |
|-----------|----------------------------------------------------------|
| PCLP-386  | <a href="#">Introducing Metrics</a>                      |
| PCLP-2285 | <a href="#">New options to control parsing limits</a>    |
| PCLP-3526 | <a href="#">Added support for AUTOSAR Rule A5-1-7</a>    |
| PCLP-3529 | <a href="#">Added support for AUTOSAR Rule A3-1-5</a>    |
| PCLP-3531 | <a href="#">Added support for AUTOSAR Rule A7-1-9</a>    |
| PCLP-3533 | <a href="#">Added support for AUTOSAR Rule A13-5-5</a>   |
| PCLP-3539 | <a href="#">Added support for AUTOSAR Rule A5-1-8</a>    |
| PCLP-3548 | <a href="#">Added support for AUTOSAR Rule A13-2-1</a>   |
| PCLP-3813 | <a href="#">Introducing Queries</a>                      |
| PCLP-3987 | <a href="#">Introduce support for new C++20 features</a> |

#### 23.3.2.2 Improvements

|           |                                                                                                                                |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|
| PCLP-1962 | <a href="#">Ignore unevaluated contexts for message 1938</a>                                                                   |
| PCLP-1977 | <a href="#">Improve performance of processing files containing heavy use of macro expansion including common Boost headers</a> |
| PCLP-2113 | <a href="#">Mark <code>std::uncaught_exception</code> as dangerous</a>                                                         |



|           |                                                                                                                                                                                                    |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2245 | Evaluate qualified calls to virtual functions with static dispatch                                                                                                                                 |
| PCLP-2606 | New <code>fit</code> flag to consider functions not declared as <code>noexcept</code> to be impure if they throw an exception                                                                      |
| PCLP-2969 | Improved support for AUTOSAR Rule A10-3-5                                                                                                                                                          |
| PCLP-2972 | Improved support for AUTOSAR Rule A12-4-2                                                                                                                                                          |
| PCLP-2997 | Deterministic field and base order for rendering of <code>class/struct/union</code> objects in Value Tracking messages                                                                             |
| PCLP-3067 | Update options for specifying language versions                                                                                                                                                    |
| PCLP-3095 | Do not emit message 9176 for dependent types                                                                                                                                                       |
| PCLP-3291 | Update documentation for message 422                                                                                                                                                               |
| PCLP-3357 | Improved analysis of constant arrays                                                                                                                                                               |
| PCLP-3511 | Support for AUTOSAR 19                                                                                                                                                                             |
| PCLP-3516 | New builtin function semantics for the <code>log</code> , <code>logf</code> , and <code>logl</code> functions                                                                                      |
| PCLP-3523 | New message 9456 supports AUTOSAR A6-5-3                                                                                                                                                           |
| PCLP-3527 | Support for AUTOSAR Rule A2-13-5                                                                                                                                                                   |
| PCLP-3528 | Support for AUTOSAR19 Rule A2-13-6                                                                                                                                                                 |
| PCLP-3532 | Message 1511 parameterized with access of hidden function                                                                                                                                          |
| PCLP-3537 | Support AUTOSAR Rule A5-1-3                                                                                                                                                                        |
| PCLP-3538 | Support for AUTOSAR A5-1-6 provided by new message 3903                                                                                                                                            |
| PCLP-3541 | Improved support for AUTOSAR Rule A13-6-1                                                                                                                                                          |
| PCLP-3594 | Improve analysis of pointers initialized in <code>if</code> statements                                                                                                                             |
| PCLP-3595 | Improve analysis of multiple return points in functions returning structures                                                                                                                       |
| PCLP-3619 | Improve handling of user-defined conversions in Value Tracking                                                                                                                                     |
| PCLP-3669 | Remove the deprecated <code>+fdu</code> flag option                                                                                                                                                |
| PCLP-3814 | Improve support for exceptions to MISRA C 2012 Rule 10.3                                                                                                                                           |
| PCLP-3816 | Extend scope of message 9034 to include switch case values                                                                                                                                         |
| PCLP-3872 | Improve text of mapped clang error and supplemental messages                                                                                                                                       |
| PCLP-3880 | Update license information for LLVM                                                                                                                                                                |
| PCLP-3901 | Update information on accommodating the <code>_bit</code> compiler extension                                                                                                                       |
| PCLP-3905 | Improved performance of <code>-d/-u</code> options appearing between <code>-env_push/-env_pop</code> options                                                                                       |
| PCLP-3912 | Improve support for MISRA C 2012 Rule 1.1 using message 793                                                                                                                                        |
| PCLP-3913 | Fix typo in appended text for CERT-C Rule POS54-C                                                                                                                                                  |
| PCLP-3919 | Add <code>walk_parent_expr</code> hook field as alias for <code>walk_pexpr</code>                                                                                                                  |
| PCLP-3920 | Remove unused suppression from MISRA C 2012 Rule 10.3                                                                                                                                              |
| PCLP-3930 | Exclude deduction guides from messages reporting function declarations                                                                                                                             |
| PCLP-3935 | Support for integer types of arbitrary size                                                                                                                                                        |
| PCLP-3955 | Improve performance for const array initialization chains when many array elements are initialized in terms of other arrays referencing further such arrays with interleaving function invocations |
| PCLP-3960 | Added support for Visual Studio 2022                                                                                                                                                               |
| PCLP-4003 | Exclude Boolean operators from MISRA C 2004 underlying type rules                                                                                                                                  |
| PCLP-4004 | Use <code>int</code> as destination underlying type for the purpose of initialization for MISRA C 2004                                                                                             |
| PCLP-4007 | Exclude initialization of pointers from MISRA C 2004 Rule 10.1                                                                                                                                     |
| PCLP-4009 | Clarify descriptions of messages 3402 and 3702                                                                                                                                                     |
| PCLP-4013 | Corrected and improved thread analysis reports                                                                                                                                                     |
| PCLP-4014 | Add Undecidable classification for MISRA C 2012 Rule 2.1                                                                                                                                           |
| PCLP-4021 | Update Frequently Asked Questions entry regarding the creation of custom messages                                                                                                                  |
| PCLP-4025 | Linux builds now require glibc 2.17                                                                                                                                                                |
| PCLP-4026 | Support the <code>char8_t</code> keyword in clang and GCC compiler configurations generated with <code>pclp_config</code>                                                                          |
| PCLP-4033 | Increase default template recursion depth limit                                                                                                                                                    |
| PCLP-4036 | Update GCC compiler configuration generation                                                                                                                                                       |
| PCLP-4041 | Update documentation with correct version of AUTOSAR targeted by <code>au-autosar.lnt</code>                                                                                                       |
| PCLP-4042 | Add missing AUTOSAR Rule M17-0-3 to <code>au-autosar.lnt</code>                                                                                                                                    |
| PCLP-4043 | Removed error messages                                                                                                                                                                             |
| PCLP-4048 | Improvements to <code>pclp_config</code>                                                                                                                                                           |

|           |                                                                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-4057 | Improved support for GCC 11 compiler configurations generated by <code>pclp_config</code>                                                 |
| PCLP-4070 | Expand support for AUTOSAR A1-1-1 and change support level to "partially supported"                                                       |
| PCLP-4073 | Remove mapping of messages previously used to provide support for AUTOSAR A4-7-1 and mark this rule as not supported                      |
| PCLP-4074 | Document limitations for support of MISRA C++ 2008 Rule 3-4-1                                                                             |
| PCLP-4075 | Parameterize <code>-append</code> options for message 829 ( <code>+headerwarn</code> option issued) used in <code>au-misra-cpp.lnt</code> |
| PCLP-4076 | Add <code>-append</code> options for message 829 ( <code>+headerwarn</code> option issued) used in <code>au-misra3-amd2.lnt</code>        |
| PCLP-4093 | Update hardware requirements to reflect uniform requirement for 64-bit CPU                                                                |
| PCLP-4094 | Parse JSON compilation databases as JSON instead of YAML within <code>pclp_config</code>                                                  |
| PCLP-4095 | Include instructions for generating a JSON compilation database with <code>iarbuild</code>                                                |
| PCLP-4100 | Improvements to the representation of symbols and types in message parameters                                                             |
| PCLP-4208 | Support multiple parsing behaviors for JSON compilation databases                                                                         |

### 23.3.2.3 Bugs Fixed

|           |                                                                                                                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2391 | Resolve false positive instance of message 2434 for dereference of pointer lexically subsequent to but excluded from control flow after an earlier conditional deallocation within a loop |
| PCLP-2859 | Resolve error 4714 for noexcept default constructor with default member initializer                                                                                                       |
| PCLP-2999 | False negative instance of message 1536 involving address of private array element                                                                                                        |
| PCLP-3190 | Resolve false negative instance of message 9011 when loop exit point count is exceeded only with consideration of a goto statement within a nested loop                                   |
| PCLP-3242 | Resolve false positive 641 involving a conditional operator expression where both the second and third operand are of the same enumeration type                                           |
| PCLP-3633 | Inappropriate suppression of messages 451, 537, and 967                                                                                                                                   |
| PCLP-3652 | Fixed crash for a friend function declaration containing an expression-form <code>noexcept</code> specifier with a default argument                                                       |
| PCLP-3855 | Resolve unexpected syntax error for structured binding to const reference with Visual Studio standard library                                                                             |
| PCLP-3864 | Resolve internal error involving switch statements containing nested control structures where all independent paths are certain to uniformly throw, continue, or exit                     |
| PCLP-3907 | Resolve false positive 727, 728, or 729 when taking the address of a parenthesized variable name                                                                                          |
| PCLP-3927 | Improve recognition of temporary output file paths when configuring TI compilers                                                                                                          |
| PCLP-3931 | Resolve internal error associated with use of deduction guides                                                                                                                            |
| PCLP-3936 | Correct domain error semantics for the <code>log1p</code> , <code>log1pf</code> , and <code>log1pl</code> functions                                                                       |
| PCLP-3959 | Message 9046 removed as supporting mechanism for AUTOSAR                                                                                                                                  |
| PCLP-3974 | False position message 2751 when thread root function is prefixed with the address-of operator                                                                                            |
| PCLP-3984 | Properly handle backslashes in the <code>command</code> field of JSON compilation databases                                                                                               |
| PCLP-4027 | Unintentionally extended comments in <code>pclp_config</code> -generated compiler configuration files                                                                                     |
| PCLP-4029 | Resolve template substitution failure for certain non-type template arguments of enumeration type                                                                                         |
| PCLP-4165 | Fix potential crash or hang when analyzing repeated definitions of a single function across multiple modules while using a large number of concurrent threads                             |
| PCLP-4188 | Resolve false negative 1540 for data members after a const data member                                                                                                                    |
| PCLP-4192 | Recognize <code>__declspec(uuid)</code>                                                                                                                                                   |
| PCLP-4200 | Recognize <code>__declspec(thread)</code>                                                                                                                                                 |

### 23.3.2.4 AUTOSAR Summary

|           |                                                          |
|-----------|----------------------------------------------------------|
| PCLP-3959 | Message 9046 removed as supporting mechanism for AUTOSAR |
| PCLP-3548 | Added support for AUTOSAR Rule A13-2-1                   |

|           |                                                                                                                      |
|-----------|----------------------------------------------------------------------------------------------------------------------|
| PCLP-3541 | Improved support for AUTOSAR Rule A13-6-1                                                                            |
| PCLP-3539 | Added support for AUTOSAR Rule A5-1-8                                                                                |
| PCLP-3538 | Support for AUTOSAR A5-1-6 provided by new message 3903                                                              |
| PCLP-3537 | Support AUTOSAR Rule A5-1-3                                                                                          |
| PCLP-3533 | Added support for AUTOSAR Rule A13-5-5                                                                               |
| PCLP-3532 | Message 1511 parameterized with access of hidden function                                                            |
| PCLP-3531 | Added support for AUTOSAR Rule A7-1-9                                                                                |
| PCLP-3529 | Added support for AUTOSAR Rule A3-1-5                                                                                |
| PCLP-3528 | Support for AUTOSAR19 Rule A2-13-6                                                                                   |
| PCLP-3527 | Support for AUTOSAR Rule A2-13-5                                                                                     |
| PCLP-3526 | Added support for AUTOSAR Rule A5-1-7                                                                                |
| PCLP-3523 | New message 9456 supports AUTOSAR A6-5-3                                                                             |
| PCLP-3511 | Support for AUTOSAR 19                                                                                               |
| PCLP-2972 | Improved support for AUTOSAR Rule A12-4-2                                                                            |
| PCLP-2969 | Improved support for AUTOSAR Rule A10-3-5                                                                            |
| PCLP-4042 | Add missing AUTOSAR Rule M17-0-3 to <code>au-autosar.lnt</code>                                                      |
| PCLP-4041 | Update documentation with correct version of AUTOSAR targeted by <code>au-autosar.lnt</code>                         |
| PCLP-4074 | Document limitations for support of MISRA C++ 2008 Rule 3-4-1                                                        |
| PCLP-4073 | Remove mapping of messages previously used to provide support for AUTOSAR A4-7-1 and mark this rule as not supported |
| PCLP-4070 | Expand support for AUTOSAR A1-1-1 and change support level to "partially supported"                                  |

#### 23.3.2.5 CERT C Summary

|           |                                                   |
|-----------|---------------------------------------------------|
| PCLP-3913 | Fix typo in appended text for CERT-C Rule POS54-C |
|-----------|---------------------------------------------------|

#### 23.3.2.6 MISRA Summary

|           |                                                                                                                                                         |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3920 | Remove unused suppression from MISRA C 2012 Rule 10.3                                                                                                   |
| PCLP-3912 | Improve support for MISRA C 2012 Rule 1.1 using message 793                                                                                             |
| PCLP-3816 | Extend scope of message 9034 to include switch case values                                                                                              |
| PCLP-3814 | Improve support for exceptions to MISRA C 2012 Rule 10.3                                                                                                |
| PCLP-3190 | Resolve false negative instance of message 9011 when loop exit point count is exceeded only with consideration of a goto statement within a nested loop |
| PCLP-3095 | Do not emit message 9176 for dependent types                                                                                                            |
| PCLP-4014 | Add Undecidable classification for MISRA C 2012 Rule 2.1                                                                                                |
| PCLP-4007 | Exclude initialization of pointers from MISRA C 2004 Rule 10.1                                                                                          |
| PCLP-4004 | Use <code>int</code> as destination underlying type for the purpose of initialization for MISRA C 2004                                                  |
| PCLP-4003 | Exclude Boolean operators from MISRA C 2004 underlying type rules                                                                                       |
| PCLP-4076 | Add <code>-append</code> options for message 829 ( <code>+headerwarn</code> option issued) used in <code>au-misra3-amd2.lnt</code>                      |
| PCLP-4075 | Parameterize <code>-append</code> options for message 829 ( <code>+headerwarn</code> option issued) used in <code>au-misra-cpp.lnt</code>               |
| PCLP-4074 | Document limitations for support of MISRA C++ 2008 Rule 3-4-1                                                                                           |

#### 23.3.3 New Features

---

|          |                                                                                                                                                                                                                              |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-386 | <b>Introducing Metrics</b><br>Metrics can be used to enforce coding guidelines and measure code quality. See the <a href="#">Metrics</a> chapter to learn how Metrics can easily enforce best practices and produce reports. |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- PCLP-2285 New options to control parsing limits**  
The new `ar_limit`, `br_limit`, `cc_limit`, and `cs_limit` options allow overriding of limits associated with operator arrow depth, nested bracket depth, `constexpr` function call depth, and maximum steps evaluated for `constexpr` functions.
- PCLP-3526 Added support for AUTOSAR Rule A5-1-7**  
Message [9426](#) is issued whenever a lambda is the operand to `typeid` or `decltype`. Complete support for AUTOSAR Rule A5-1-7 is now provided.
- PCLP-3529 Added support for AUTOSAR Rule A3-1-5**  
Message [9449](#) is issued whenever a non-template member function is defined within its class definition. Support for AUTOSAR Rule A3-1-5 is now provided.
- PCLP-3531 Added support for AUTOSAR Rule A7-1-9**  
Message [9428](#) is issued whenever an `enum`, `union`, `struct`, or `class` type is defined in the same statement with the declaration of a variable of its type. Complete support for AUTOSAR Rule A7-1-9 is now provided.
- PCLP-3533 Added support for AUTOSAR Rule A13-5-5**  
These messages apply only to user-declared comparison operator functions (`operator==`, `operator!=`, `operator<`, `operator<=`, `operator>=`, and `operator>`). Message [9444](#) is issued whenever the function is a `class/struct` member. Message [9445](#) is issued whenever the function is not `noexcept` (`noexcept(true)`, `throw()`, or `__declspec(nothrow)`). Message [9446](#) is issued whenever the parameters to the function have different types. Support for AUTOSAR Rule A13-5-5 is now provided.
- PCLP-3539 Added support for AUTOSAR Rule A5-1-8**  
Message [9442](#) is issued whenever a lambda definition is nested inside another lambda definition. Support for AUTOSAR Rule A5-1-8 is now provided.
- PCLP-3548 Added support for AUTOSAR Rule A13-2-1**  
Messages [9447](#) is issued whenever the return type of a user defined assignment operator is not a reference to the left hand parameter ('this' for member functions and the first parameter for non-member functions) with the same `const/volatile` qualification. Message [9448](#) is issued whenever the return value of a user defined assignment operator is not a reference to the left hand parameter ('\*this' for member functions and the first parameter for non-member functions) with the same `const/volatile` qualification. Support for AUTOSAR Rule A13-2-1 is now provided.
- PCLP-3813 Introducing Queries**  
The new [PC-lint Plus Query Language](#) provides a facility to extend the functionality of PC-lint Plus by defining custom checks and diagnostics and suppressing messages based on dynamic introspection.
- PCLP-3987 Introduce support for new C++20 features**  
PC-lint Plus 2.0 introduces support for many C++20 features including three-way comparison (`operator<=>`), extended `constexpr`, and partial support for Concepts and Coroutines. The C++20 Modules feature is not currently supported.

### 23.3.4 Improvements

- 
- PCLP-1962    **Ignore unevaluated contexts for message 1938**  
 Message 1938 is no longer emitted for references to global variables within unevaluated contexts such as `sizeof` expressions.
- PCLP-1977    **Improve performance of processing files containing heavy use of macro expansion including common Boost headers**  
 Performance has been improved in situations where many large, nested macro expansions are encountered. This particularly improves processing speed when including a library such as Boost Phoenix.
- PCLP-2113    **Mark `std::uncaught_exception` as dangerous**  
 Usage of the former standard library function `std::uncaught_exception` will now elicit message 421. This function has been replaced by `std::uncaught_exceptions`.
- PCLP-2245    **Evaluate qualified calls to virtual functions with static dispatch**  
 Value Tracking now walks the static target function when evaluating a qualified call to a virtual function.
- PCLP-2606    **New `fit` flag to consider functions not declared as `noexcept` to be impure if they throw an exception**  
 The new `fit` flag is off by default. If the flag is turned on (such as with the option `+fit`), then a `throw` expression within a function body will render that function impure unless the function is declared as not throwing any exceptions (in which case a thrown exception cannot be allowed to escape by any control path).
- PCLP-2969    **Improved support for AUTOSAR Rule A10-3-5**  
 Message 9438 will report virtual non-copy/non-move assignment operator functions. With messages 9407 and 9410, complete support for AUTOSAR Rule A10-3-5 is now provided.
- PCLP-2972    **Improved support for AUTOSAR Rule A12-4-2**  
 Message 9441 will report public non-virtual destructors in effectively non-final classes. A class is effectively final if either it or its destructor is final. Complete support for AUTOSAR Rule A2-13-5 is now provided.
- PCLP-2997    **Deterministic field and base order for rendering of `class/struct/union` objects in Value Tracking messages**  
 The order in which fields and bases of an object were displayed within a particular instance of a Value Tracking message could previously vary between repeated runs of PC-lint Plus. These items are now deterministically displayed in alphabetical order. The existing behavior of listing all bases first and all fields second still applies, and as such the bases and fields are alphabetized separately among themselves.
- PCLP-3067    **Update options for specifying language versions**  
 The `-std` option now accepts `-std=c++20` to specify C++20. The previously deprecated `-A` option is no longer listed in the documentation. The legacy `-A` option was considered deprecated in the initial release of PC-lint Plus version 1.0; this option is now obsolete. The `-std` option should be used instead.

- PCLP-3095 Do not emit message 9176 for dependent types**  
 Message 9176 is no longer emitted for casts involving dependent types within templates. Casts arising from specific template instantiations will still be reported. For example,
- ```
//lint -w1 +e9176

template<typename T, typename U>
void f() {
    T* t = 0;
    reinterpret_cast<U*>(t);
}

void g() {
    f<int, double>();
}
```
- previously emitted two instances of message 9176, one for the generic cast from `T*` to `U*` and a second for the specific cast from `int*` to `double*`. Message 9176 now only reports the latter.
- PCLP-3291 Update documentation for message 422**  
 The documentation for message 422 has been updated to remove an inapplicable example and clarify the circumstances in which the message is emitted.
- PCLP-3357 Improved analysis of constant arrays**  
 The specific values of array elements of constant scalar arrays are now used to support the analysis of expressions that reference them.
- PCLP-3511 Support for AUTOSAR 19**  
 PC-lint Plus previously provided support only for AUTOSAR 17 coding guidelines. Support is now provided for AUTOSAR 19 coding guidelines as well (the latest version of the AUTOSAR standard).
- PCLP-3516 New builtin function semantics for the log, logf, and logl functions**  
 PC-lint Plus will now detect potential domain errors resulting from misuse of the `log`, `logf`, and `logl` Standard C library functions.
- PCLP-3523 New message 9456 supports AUTOSAR A6-5-3**  
 The new message 9456 now reports on a `do` statement that is used outside of a statement-like macro definition.
- PCLP-3527 Support for AUTOSAR Rule A2-13-5**  
 Message 9439 will report hexadecimal floating and integer literals that contain lower case digits (a-f). Complete support for AUTOSAR Rule A2-13-5 is now provided.
- PCLP-3528 Support for AUTOSAR19 Rule A2-13-6**  
 Message 9423 (non-basic character used in identifier) can also be used to indicate identifiers that use non-basic characters.  
 AUTOSAR19 Rule A2-13-6 is now supported by the new messages 9443 (universal-character-name used in identifier).
- PCLP-3532 Message 1511 parameterized with access of hidden function**  
 Message 1511 (member function hides non-virtual member) is now parameterized by the member access of the function being hidden allowing suppression of the message for e.g. hidden private members (using `-estring(1511,private)`). This change supports the exception to AUTOSAR19 Rule A10-2-1 for private member functions.

- PCLP-3537     **Support AUTOSAR Rule A5-1-3**  
AUTOSAR Rule A5-1-3 is now supported by the new message **9424** (parameter list omitted from lambda expression).
- PCLP-3538     **Support for AUTOSAR A5-1-6 provided by new message 3903**  
The new message **3903** provides support for AUTOSAR A5-1-6 by reporting lambda expressions without explicitly specified return types.
- PCLP-3541     **Improved support for AUTOSAR Rule A13-6-1**  
Message **9440** is issued whenever the digit sequence separators (') in an integral or floating literal do not occur every Nth digit, where N is: 2 for hexadecimal, 3 for decimal and octal, and 4 for binary. Complete support for AUTOSAR Rule A13-6-1 is now provided.
- PCLP-3594     **Improve analysis of pointers initialized in if statements**  
The existing support for analysis of modifications to a pointer value made within if statements has been extended to improve the handling of cases where a pointer first obtains a meaningful value when values are assigned in both paths through an if statement.
- PCLP-3595     **Improve analysis of multiple return points in functions returning structures**  
The analysis of possible return values associated with various conditional return points within a called function now has improved support for merging values within returned structures.
- PCLP-3619     **Improve handling of user-defined conversions in Value Tracking**  
The circumstances under which Value Tracking evaluates implicit calls to user-defined conversion operators have been generalized to improve analysis quality.
- PCLP-3669     **Remove the deprecated +fdu flag option**  
The fdu flag option was previously deprecated in PCLP-3514 and has now been fully removed.
- PCLP-3814     **Improve support for exceptions to MISRA C 2012 Rule 10.3**  
Message 9034 now implements an additional exception for initialization with 0 to improve support for MISRA C 2012 Rule 10.3.
- PCLP-3816     **Extend scope of message 9034 to include switch case values**  
Message 9034 now applies MISRA C 2012 Rule 10.3 to the conversion of a switch case value to the type of the switch condition expression.
- PCLP-3872     **Improve text of mapped clang error and supplemental messages**  
References to clang options which do not apply to PC-lint Plus have been removed from the text of various mapped clang messages for clarity.
- PCLP-3880     **Update license information for LLVM**  
The Open Source Declarations have been updated to reflect the new license employed by the LLVM project.
- PCLP-3901     **Update information on accommodating the \_bit compiler extension**  
The documentation no longer recommends the use of reserved word options for the `_bit` keyword which some compilers use to name a one-bit integral type. See PCLP-3935.



**PCLP-3905 Improved performance of -d/-u options appearing between -env\_push/-env\_pop options**

Project configurations generated with the `pclp_config` utility typically enclose each module in a block that begins with `-env_push` and ends with a corresponding `-env_pop`. Within this block are relevant `-d` options that define macros the correspond to similar compiler options. When the total number of such options across all blocks for a project was very large, severe performance degradation could result. This issue has been corrected.

**PCLP-3912 Improve support for MISRA C 2012 Rule 1.1 using message 793**

Message [793](#) expands support for MISRA C 2012 Rule 1.1 by reporting when translation limits are exceeded.

**PCLP-3913 Fix typo in appended text for CERT-C Rule POS54-C**

The appended text for violations of POS54-C reported by message [534](#) when using `au-certc.lnt` now correctly refers to POS54-C where it previously referenced POS53-C.

**PCLP-3919 Add walk\_parent\_expr hook field as alias for walk\_pexpr**

The `walk_parent_expr` hook field documented in the Hooks Programmer's Guide was previously only accessible as `walk_pexpr`. This field is now also accessible with the name `walk_parent_expr`.

**PCLP-3920 Remove unused suppression from MISRA C 2012 Rule 10.3**

An `-estring` option for message [9052](#) has been removed from the Rule 10.3 section of the MISRA C 2012 configuration file. This suppression was not applicable as message 9052 is not enabled by the MISRA C 2012 configuration file and is not used to provide support for any MISRA C 2012 rules.

**PCLP-3930 Exclude deduction guides from messages reporting function declarations**

Messages concerning function declarations such as [2701](#), [902](#) and [955](#) have been updated to prevent inapplicable messages for deduction guides. Note that message [9141](#) will still report deduction guides.

**PCLP-3935 Support for integer types of arbitrary size**

PC-lint Plus now supports the `_ExtInt` extension which allows support for arbitrarily-sized integers, including single-bit types.

**PCLP-3955 Improve performance for const array initialization chains when many array elements are initialized in terms of other arrays referencing further such arrays with interleaving function invocations**

Performance has been improved when processing larger examples of the form:

```
int g();

int const          a[] = { 1+g(), 2+g(), 3+g() };
int const * const  b[] = { a+g(), a+g(), a+g() };
int const * const * const c[] = { b+g(), b+g(), b+g() };

void f() {
    c;
}
```

which previously exhibited performance differences based on the presence or absence of `const` qualification.



- PCLP-3960 Added support for Visual Studio 2022**  
The automated configuration tool (`pclp_config.py`) has been enhanced to support Visual Studio 2022 and later versions of Visual Studio 2019.  
The C++ feature test macros in the generated compiler configuration header files are now filtered to match what PC-lint Plus supports. A new option (`--dont-filter-feature-test-macros`) has been added to disable this filtering if required.
- PCLP-4003 Exclude Boolean operators from MISRA C 2004 underlying type rules**  
While Boolean operators yield a result of type `int`, MISRA has clarified that they do not intend for them to participate in underlying type enforcement. Expressions classified as Boolean-by-Construction are now excluded from underlying type classification.
- PCLP-4004 Use `int` as destination underlying type for the purpose of initialization for MISRA C 2004**  
Initialization of a variable of enumeration type could previously result in an instance of message [9225](#) reporting an underlying type conversion with an enumeration type as the destination type. In such cases the destination type used in the message and for the purpose of determining whether the message is issued will now be `int`.
- PCLP-4007 Exclude initialization of pointers from MISRA C 2004 Rule 10.1**  
Violations of MISRA C 2004 Rule 10.1 will no longer be reported when an integer, e.g. a null pointer constant, is used to initialize a pointer.
- PCLP-4009 Clarify descriptions of messages [3402](#) and [3702](#)**  
The descriptions of messages [3402](#) and [3702](#) have been updated to clarify that they are emitted for lambdas that default to capture by value and by reference, respectively.
- PCLP-4013 Corrected and improved thread analysis reports**  
The generated machine readable (CSV, JSON, and XML) thread analysis reports could be invalid. All known issues with these reports have been corrected. The layout of the (text format) threads report has been improved by placing the thread name on its own line and splitting the root function prototype and its definition source line information on separate lines. In addition, several new status fields associated with the report filters have been added.
- PCLP-4014 Add Undecidable classification for MISRA C 2012 Rule 2.1**  
The MISRA C 2012 support matrix now shows rule 2.1 as "Req\*" rather than "Req", indicating that it is undecidable.
- PCLP-4021 Update Frequently Asked Questions entry regarding the creation of custom messages**  
The Frequently Asked Question entry about creating new messages now notes that the new Queries feature provides this capability.
- PCLP-4025 Linux builds now require glibc 2.17**  
The required version of glibc has increased from 2.11 to 2.17.
- PCLP-4026 Support the `char8_t` keyword in clang and GCC compiler configurations generated with `pclp_config`**  
Clang and GCC compiler configurations generated by `pclp_config` will now enable the `char8_t` keyword when appropriate.

- PCLP-4033 Increase default template recursion depth limit**  
The default template recursion depth limit (modifiable via the `-tr_limit` option) has increased from 75 to 1024.
- PCLP-4036 Update GCC compiler configuration generation**  
Configurations generated for GCC previously contained the options `--u_GLIBCXX_BUILTIN_IS_SAME_AS` and `--u_GLIBCXX_HAVE_BUILTIN_IS_CONSTANT_EVALUATED` to prevent the use of unavailable extended keywords. These keywords are now supported and these options are no longer generated.
- PCLP-4041 Update documentation with correct version of AUTOSAR targeted by `au-autosar.lnt`**  
The documentation for Beta 1 stated that the provided `au-autosar.lnt` file targeted AUTOSAR 17-10, the actual targeted version is AUTOSAR 17-03. The documentation has been corrected.
- PCLP-4042 Add missing AUTOSAR Rule M17-0-3 to `au-autosar.lnt`**  
An entry for the (unsupported) AUTOSAR Rule M17-0-3 has been added to `au-autosar.lnt`.
- PCLP-4043 Removed error messages**  
The following errors have been removed from PC-lint Plus: 64 (incompatible types in initialization), 111 (assignment to const object), 1005 (destructor declaration requires a class), 1049 (template argument arity mismatch), 1050 (missing template args), 1111 (illegal explicit specialization), 1121 (no viable function for range expression), and 1122 (range expression begin/end mismatch). PC-lint Plus will diagnose relevant errors with a mapped clang error in the 4001-6999 message range.
- PCLP-4048 Improvements to `pclp_config`**  
Compiler configuration files generated by `pclp_config.py` now contain additional information in the introductory comment that further document when and how they were generated. The `--compiler-c-options` options or `--compiler-cpp-options` options will be applied whenever the compiler is invoked. Previously these options were only applied when the compiler was invoked to obtain the predefined macros. Additional debugging options were added. For more details, see the `pclp_config` Options Reference.
- PCLP-4057 Improved support for GCC 11 compiler configurations generated by `pclp_config`**  
Compiler configurations generated by `pclp_config` for GCC or clang compilers will now include options that prevent the use of unsupported GCC features by standard library headers via the `__attr_dealloc` and `__attr_dealloc_free` macros.
- PCLP-4070 Expand support for AUTOSAR A1-1-1 and change support level to "partially supported"**  
The detection of deprecated C++ standard library features supporting AUTOSAR A1-1-1 has been expanded. The support status for this guideline has been changed to "partially supported" and the scope of support is now described in the support matrix.
- PCLP-4073 Remove mapping of messages previously used to provide support for AUTOSAR A4-7-1 and mark this rule as not supported**  
Messages 9119 and 9120 are no longer associated with AUTOSAR A4-7-1, and this rule is now categorized as not supported.

- PCLP-4074     **Document limitations for support of MISRA C++ 2008 Rule 3-4-1**  
 Message 9003 (could define global variable within function) is used to support MISRA C++ 2008 Rule 3-4-1 (and the corresponding AUTOSAR Rule M3-4-1) but has limitations that prevent it from providing full coverage for this rule. The support status of this rule has been downgraded to "partially supported" with the limitations noted in the support matrix.
- PCLP-4075     **Parameterize `-append` options for message 829 (`+headerwarn` option issued) used in `au-misra-cpp.lnt`**  
 The `+headerwarn` option is used to support MISRA C++ 2008 Rules 18-0-1, 18-0-4, 18-7-1, and 27-0-1 in the `au-misra-cpp.lnt` file but the corresponding `-append` options were not parameterized with the names of the offending headers. The result was that use of a header that was prohibited by any of these guidelines was diagnosed as a violation of all of them. This issue has been corrected and only the appropriate guideline will now be referenced in the violation message.
- PCLP-4076     **Add `-append` options for message 829 (`+headerwarn` option issued) used in `au-misra3-amd2.lnt`**  
 The `+headerwarn` options is used to support MISRA C 2012 Rule 1.4 in the `au-misra3-amd2.lnt` file did not contain corresponding `-append` options correlating the resulting issuance of message 829 (`+headerwarn` option issued) to the violated rule. This issue has been corrected, and message 829 issued as a result of using `stdalign.h` or `stdnoreturn.h` will now reference the corresponding MISRA guideline.
- PCLP-4093     **Update hardware requirements to reflect uniform requirement for 64-bit CPU**  
 The hardware requirements have been updated to reflect that PC-lint Plus 2.0 requires a 64-bit CPU and is not distributed with 32-bit binaries for any platform.
- PCLP-4094     **Parse JSON compilation databases as JSON instead of YAML within `pclp_config`**  
 The `pclp_config` utility previously used the `yaml` Python module to parse JSON compilation databases when generating project configurations which could result in a parse error when tabs were used for indentation as occurs in compilation databases produced by `iarbuild`. The `json` module is now used to parse JSON compilation databases by `pclp_config`.
- PCLP-4095     **Include instructions for generating a JSON compilation database with `iarbuild`**  
 Instructions for generating a JSON compilation database with recent versions of `iarbuild` using the `-jsondb` option are now provided.
- PCLP-4100     **Improvements to the representation of symbols and types in message parameters**  
 Various minor improvements to the general representation of symbols and types appearing in message parameterizations including:  
 - The Boolean type is represented as `bool` instead of `_Bool` in C++ mode.  
 - The `restrict` qualifier is consistently represented in C mode even when written using an alternate keyword such as `__restrict`.  
 - Defaulted template type parameters are elided from the representation of the instantiated type.  
 - The representation of nested template types elides spaces between adjacent closing angle brackets in C++11 and later.

- PCLP-4208     **Support multiple parsing behaviors for JSON compilation databases**  
 The parsing behavior used when `pclp_config.py` processes a JSON compilation database now defaults to posix-like for Linux and macOS and non-posix-like for Windows. The new options `--posix-command-parsing` and `--no-posix-command-parsing` may be used to override the default behavior. The parsing behavior affects how backslashes and quotes are handled in response files and arguments embedded in the command field of JSON compilation databases.

### 23.3.5 Bugs Fixed

- 
- PCLP-2391     **Resolve false positive instance of message 2434 for dereference of pointer lexically subsequent to but excluded from control flow after an earlier conditional deallocation within a loop**  
 A false positive instance of message 2434 could have previously occurred when, within a loop, a pointer was dereferenced subsequent to an `if` statement that conditionally executed a sequence consisting of the deallocation of the memory associated with the target of the pointer and an unconditional modification of control flow rendering the report site and the deallocation mutually exclusive. This issue has been resolved.
- PCLP-2859     **Resolve error 4714 for noexcept default constructor with default member initializer**  
 A spurious instance of error 4714 (default member initializer needed within enclosing class) for a noexcept default constructor has been resolved.
- PCLP-2999     **False negative instance of message 1536 involving address of private array element**  
 An issue in which an expected instance of message 1536 may not have been emitted when a member function returned the result of applying the address-of operator to a specific element of a private array data member has been resolved.
- PCLP-3190     **Resolve false negative instance of message 9011 when loop exit point count is exceeded only with consideration of a goto statement within a nested loop**  
 An issue in which message 9011 may not appear when the presence of multiple exits is dependent on the attribution of a transfer of control within a nested loop to an outer loop has been resolved.
- PCLP-3242     **Resolve false positive 641 involving a conditional operator expression where both the second and third operand are of the same enumeration type**  
 A conditional operator expression whose second and third operands are of the same enumeration type is now treated as resulting in a value of the common enumeration type. This resolves false positive instances of message 641 that could previously occur when an operation combined a recognized enumeration value and an unrecognized enumeration value from a conditional operator.
- PCLP-3633     **Inappropriate suppression of messages 451, 537, and 967**  
 An `#include` directive with a file name that contained a directory separator was previously treated as library for the purposes of messages 451, 537, 967 which could result in a false negative in situations where the header was not a library file and the corresponding message was enabled for non-library files but disabled for library files. Additionally, messages 451 and 537 were previously unintentionally treated as appearing in a library region when the file being included was a library header but the `#include` directive appeared in a non-library region. These issues have been corrected.

- PCLP-3652 Fixed crash for a friend function declaration containing an expression-form `noexcept` specifier with a default argument**  
 PC-lint Plus would previously crash while parsing a friend function declaration that contained both a default argument and an expression-form `noexcept` specifier. This was most notably encountered when using the ASIO library which required using either the `++dASIO_NOEXCEPT_IF(x)=` or `++dBOOST_ASIO_NOEXCEPT_IF(x)=` options as a work-around. The underlying issue has been resolved and the previously-provided work-arounds are no longer necessary.
- PCLP-3855 Resolve unexpected syntax error for structured binding to `const` reference with Visual Studio standard library**  
 An unexpected syntax error that could occur when a structured binding attempted to bind to a `const` reference when using the Microsoft Visual Studio standard library has been resolved.
- PCLP-3864 Resolve internal error involving switch statements containing nested control structures where all independent paths are certain to uniformly throw, continue, or exit**  
**PCLP-3921**  
 An internal error with error code 197D39F9 and diagnostic info -37 that could occur in specific situations involving switch statements containing nested control structures where all independent paths are certain to uniformly throw, continue, or exit has been resolved.
- PCLP-3907 Resolve false positive 727, 728, or 729 when taking the address of a parenthesized variable name**  
 False positive instances of messages 727, 728, or 729 that could have previously occurred when the operand to the address-of operator was the name of a variable enclosed in parentheses have been resolved.
- PCLP-3927 Improve recognition of temporary output file paths when configuring TI compilers**  
 An issue that could prevent the automated detection of macro definitions and size options from TI compilers when temporary files were produced in certain directories has been resolved.
- PCLP-3931 Resolve internal error associated with use of deduction guides**  
 An internal error with the Clang message “Can’t mangle a deduction guide name!” when a constructor call uses a deduction guide has been resolved.
- PCLP-3936 Correct domain error semantics for the `log1p`, `log1pf`, and `log1pl` functions**  
 A false positive 2423 (apparent domain error for function) or 2623 (possible domain error for function) message could be emitted when the `log1p`, `log1pf`, and `log1pl` functions were called with an argument having a value of less than one instead of a value of less than negative one. This issue has been corrected.
- PCLP-3959 Message 9046 removed as supporting mechanism for AUTOSAR**  
 References to message 9046 (symbol is typographically ambiguous) have been removed from the autosar author files as this message is not currently supported for C++ modules.
- PCLP-3974 False position message 2751 when thread root function is prefixed with the address-of operator**  
 If the thread root function passed to the `std::thread` constructor is prefixed with the address-of operator a false positive message 2751 (indeterminable thread root function) was previously emitted. This issue is now resolved.

- PCLP-3984 Properly handle backslashes in the `command` field of JSON compilation databases**  
 When directory separators are represented using backslashes in the `command` field of a JSON compilation database, the resulting project configuration generated by `pclp_config.py` would not include the directory separators. This situation could occur when using `cmake` with a Makefile generator on Windows. This issue has been resolved.
- PCLP-4027 Unintentionally extended comments in `pclp_config`-generated compiler configuration files**  
 When a compiler option specified via one of `--compiler-options`, `--compiler-c-options`, or `--compiler-cpp-options` ended with a backslash, the explanatory C++-style comment that followed the transformed option could also end with a backslash eliciting message 427 (comment continued via back-slash) and causing any options on the following line to be ignored. For example, using:  

```
--compiler-options=-I\
```

  
 could result in the following line generated in the produced configuration file:  

```
-i\" // From compiler option(s): -I\
```

  
 Compiler options are now enclosed in single quotes within the explanatory comment.
- PCLP-4029 Resolve template substitution failure for certain non-type template arguments of enumeration type**  
 An error most likely to manifest through message 5819 where template substitution failed due to the forwarding of the value of a non-type template parameter of enumeration type from an outer class template to a template argument list appearing within the instantiation of an inner member function or constructor template has been resolved.
- PCLP-4165 Fix potential crash or hang when analyzing repeated definitions of a single function across multiple modules while using a large number of concurrent threads**  
 PCLP-3630  
 An issue that could cause a crash or hang during global wrap-up or a later pass when analyzing a large project containing multiple definitions of the same function in many different modules while utilizing a large number of concurrent threads has been resolved.
- PCLP-4188 Resolve false negative 1540 for data members after a `const` data member**  
 PCLP-3956  
 Message 1540 previously exhibited false negatives for data members appearing subsequent to a data member declared `const`. This issue has been resolved.
- PCLP-4192 Recognize `__declspec(uuid)`**  
 PC-lint Plus 2.0 beta versions incorrectly emitted message (5957) for all calls to `__uuidof` because it did not correctly recognize the Microsoft `uuid` attribute. This regression error has been corrected.
- PCLP-4200 Recognize `__declspec(thread)`**  
 The 2.0 beta versions silently ignored `__declspec(thread)` instead of treating it as equivalent to `thread_local`. This regression error has been corrected.

### 23.3.6 Known Issues

- PCLP-939** [Message 756 is disabled](#)

|           |                                                                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-960  | Message 9103 not issued for inter-module reuse                                                                                                |
| PCLP-2135 | False negative 9003 for static variables                                                                                                      |
| PCLP-2603 | Message 767 is disabled                                                                                                                       |
| PCLP-2614 | False positive instance of message 522 for call to const member function but has external side-effects                                        |
| PCLP-3004 | Message 1565 does not report the status of sub-objects within data members of class type                                                      |
| PCLP-3197 | False negative instance of message 9034 involving compound assignment                                                                         |
| PCLP-3198 | False positive 9045 for nested structures                                                                                                     |
| PCLP-3213 | False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter |
| PCLP-3223 | The undocumented <code>+fil</code> flag is not implemented                                                                                    |
| PCLP-3569 | False positive 550 for variable initializing const reference member using braced syntax                                                       |
| PCLP-3660 | False positive instance of message 776 for addition between int and long of equal size                                                        |
| PCLP-3803 | False positive 650 for shifted values                                                                                                         |
| PCLP-3856 | False positive side-effect related messages for a lambda that has side effects                                                                |
| PCLP-4010 | False positive 9049 within lambda body                                                                                                        |
| PCLP-4046 | False positive 1764 for universal reference parameter of template instantiated with a lambda                                                  |

---

|           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-939  | <b>Message 756 is disabled</b><br>Message <a href="#">756</a> (global typedef not referenced) is disabled in the current release.                                                                                                                                                                                                                                                                                                                                     |
| PCLP-960  | <b>Message 9103 not issued for inter-module reuse</b><br>Message <a href="#">9103</a> (identifier with static storage reused) is not issued when a reuse of a static storage identifier occurs in a separate module.                                                                                                                                                                                                                                                  |
| PCLP-2135 | <b>False negative 9003 for static variables</b><br>A file-scope static variable referenced only within a single function will not be reported by <a href="#">9003</a> . The message will be issued for variables that are implicitly or explicitly <code>extern</code> .                                                                                                                                                                                              |
| PCLP-2603 | <b>Message 767 is disabled</b><br>Message <a href="#">767</a> is disabled in the current release.                                                                                                                                                                                                                                                                                                                                                                     |
| PCLP-2614 | <b>False positive instance of message 522 for call to const member function but has external side-effects</b><br>A false positive instance of message <a href="#">522</a> may be emitted for a call to a const member function despite the presence of external side-effects.                                                                                                                                                                                         |
| PCLP-3004 | <b>Message 1565 does not report the status of sub-objects within data members of class type</b><br>Message <a href="#">1565</a> will report when an initializer function has not initialized a non-static data member of the class of which the initializer function is a non-static member function. It will not report based on the status of non-static data members of other objects nested recursively as sub-objects of a non-static data member of class type. |
| PCLP-3197 | <b>False negative instance of message 9034 involving compound assignment</b><br>A compound assignment involving operands of different essential type categories may be reported by message <a href="#">9029</a> , but compound assignment may not elicit an expected instance of message <a href="#">9034</a> .                                                                                                                                                       |



- PCLP-3198 **False positive 9045 for nested structures**  
 Message [9045](#) (complete definition of symbol is unnecessary in this translation unit) may be incorrectly issued for structures whose only use appears within another structure.
- PCLP-3213 **False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter**  
 When providing a cast expression as a template argument to a non-type template parameter some messages described as reporting implicit conversions may be issued as if the cast operand were implicitly converted to the cast type.
- PCLP-3223 **The undocumented +fil flag is not implemented**  
 The +fil flag, which would control whether or not indentation checking is applied to labels, is not implemented. This unimplemented flag is not documented in the list of flag options, but a reference to it is made in section 13.4.
- PCLP-3569 **False positive 550 for variable initializing const reference member using braced syntax**  
 Message [550](#) (local variable not subsequently accessed) may be incorrectly issued when the only use of the referenced variable is to initialize a const-qualified reference member in the member-init-list of a constructor using braced syntax.
- PCLP-3660 **False positive instance of message 776 for addition between int and long of equal size**  
 The addition of an `int` and `long` while using a configuration where `sizeof(int) == sizeof(long)` may result in a false positive instance of message [776](#).
- PCLP-3803 **False positive 650 for shifted values**  
 Message [650](#) (constant out of range for operator) is sometimes incorrectly issued when the result of a shift expression is compared with a constant value.
- PCLP-3856 **False positive side-effect related messages for a lambda that has side effects**  
 Side effects resulting from the invocation of a closure type's function call operator may elicit diagnostics (including [522](#), [523](#), and [438](#)) that incorrectly imply the lack of a corresponding side effect.
- PCLP-4010 **False positive 9049 within lambda body**  
 Message [9049](#) (increment/decrement operator used in expression with other side-effects) is incorrectly issued for increment and decrement expressions appearing as part of a statement without additional side-effects within the body of a lambda expression.
- PCLP-4046 **False positive 1764 for universal reference parameter of template instantiated with a lambda**  
 Message [1764](#) (reference parameter of function could be reference to const) may be incorrectly issued when a function template is instantiated with a lambda as the argument corresponding to a universal reference parameter.

### 23.3.7 Changes to Beta Functionality Since Version 2.0 Beta 3

- PCLP-4190 [Resolve false positive messages involving conversions of integer constant expressions](#)  
 PCLP-4198 [Resolve incorrect library classification for file metrics](#)  
 PCLP-4207 [Correct undefined variable regression error within `pclp\_config`](#)



---

|           |                                                                                                                                                                                                                                                                                                                                                                  |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-4190 | <b>Resolve false positive messages involving conversions of integer constant expressions</b><br>False positive instances of messages involving the conversion of integer constant expressions such as message 569 in previous betas of version 2.0 have been resolved.                                                                                           |
| PCLP-4198 | <b>Resolve incorrect library classification for file metrics</b><br>Library status classification for file metric hosts was introduced in beta 2, but an issue that affected this classification could cause library files to be unexpectedly included in (when -flo is off) or excluded from (when +flo is on) the metric report. This issue has been resolved. |
| PCLP-4207 | <b>Correct undefined variable regression error within pclp_config</b><br>The 2.0 beta 3 version of pclp_config would abort if it was unable to determine the version information of the selected compiler. This issue has been corrected.                                                                                                                        |

## 23.4 Version 1.4.1

### 23.4.1 Summary

#### 23.4.1.1 Bugs Fixed

|           |                                                                                                                                                                                                       |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3802 | Resolve internal error associated with precision messages                                                                                                                                             |
| PCLP-3818 | Resolve false positives related to the use of <b>throw</b> , <b>continue</b> , and <b>return</b> within <b>switch</b> statements                                                                      |
| PCLP-3822 | Resolve false negatives in non-library code appearing subsequent to an include directive for a library header and prior to any intervening lint comment or include directive for a non-library header |

## 23.5 Version 1.4

### 23.5.1 Summary

#### 23.5.1.1 New Features

|           |                                                                                                                              |
|-----------|------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2033 | Add message 2418 to report declarations with conflicting strong types                                                        |
| PCLP-2440 | Detect and diagnose multi-threading issues                                                                                   |
| PCLP-2744 | Added support for CERT recommendation CON01-C                                                                                |
| PCLP-2753 | Detect dangerous accesses between threads                                                                                    |
| PCLP-2757 | Detect inconsistent mutex locking order                                                                                      |
| PCLP-2762 | Detect dangerous accesses between threads                                                                                    |
| PCLP-3022 | New message 2415 reports out of range comparisons in loop conditions                                                         |
| PCLP-3550 | New builtin <b>__lint_assert</b> function and <b>assert</b> macro override for pclp_config-generated compiler configurations |
| PCLP-3709 | Automatic compiler configuration generation for Texas Instruments Code Composer Studio compilers using pclp_config.py        |

#### 23.5.1.2 Improvements

|           |                                                                                                             |
|-----------|-------------------------------------------------------------------------------------------------------------|
| PCLP-805  | JSON Compilation Database support for pclp_config                                                           |
| PCLP-1662 | Messages 655 and 656 now issued with a <i>type</i> parameter, portrayal of tagless enumerations is improved |
| PCLP-2072 | Message 1520 no longer issued for non-copy non-move assignment operators                                    |
| PCLP-2299 | Improve tracking of active member changes for unions containing structures                                  |
| PCLP-2331 | New options to control modules included in project configurations generated with pclp_config                |

|           |                                                                                                                                                                                                                      |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2569 | Improved error handling for the <code>-sem</code> option                                                                                                                                                             |
| PCLP-2589 | Message 9035 will now report any use of VLAs                                                                                                                                                                         |
| PCLP-2681 | Slash equivalency in filename patterns                                                                                                                                                                               |
| PCLP-2891 | Member hiding messages are no longer used to support MISRA C++ Rule 2-10-2                                                                                                                                           |
| PCLP-3013 | Message 1511 no longer reports assignment operators                                                                                                                                                                  |
| PCLP-3087 | Improved warnings for common misuse of the <code>-i</code> option                                                                                                                                                    |
| PCLP-3134 | Messages 655 and 656 now supported for C++ modules                                                                                                                                                                   |
| PCLP-3141 | Clarify scope of message 565                                                                                                                                                                                         |
| PCLP-3192 | Improvements to message 9020 (header file name with non-standard character sequence)                                                                                                                                 |
| PCLP-3204 | Message 9137 is now deprecated                                                                                                                                                                                       |
| PCLP-3219 | Compound assignment considered a side effect for messages 666 and 2666                                                                                                                                               |
| PCLP-3260 | Improvements to for statement messages 440, 442, 443, and 444                                                                                                                                                        |
| PCLP-3422 | Improved reporting of conversions appearing in C++ constructors.                                                                                                                                                     |
| PCLP-3491 | Improved precision handling of enumerations without enumerators                                                                                                                                                      |
| PCLP-3508 | Improved reporting of conversions appearing in initializer lists                                                                                                                                                     |
| PCLP-3514 | Deprecation of the <code>fdi</code> flag option                                                                                                                                                                      |
| PCLP-3546 | Improved support for AUTOSAR Rule A18-0-3                                                                                                                                                                            |
| PCLP-3558 | Disassociate inappropriate messages from CERT C guideline MSC12-C                                                                                                                                                    |
| PCLP-3562 | Exclude operands of Boolean and enumeration type when checking BARR-C:2018 Rule 5.3c                                                                                                                                 |
| PCLP-3566 | Use <code>/usr/bin/env</code> to select the interpreter for <code>pclp_config</code>                                                                                                                                 |
| PCLP-3567 | Changes to handling of <code>#pragma once</code> and alternate include guards for 451 and 967                                                                                                                        |
| PCLP-3572 | Improve Value Tracking of switch statements                                                                                                                                                                          |
| PCLP-3596 | Improved library determination of symbols reported in global wrap-up messages                                                                                                                                        |
| PCLP-3597 | Improved handling of project include directories for configurations generated with <code>pclp_config</code>                                                                                                          |
| PCLP-3598 | Improved portrayal of permanent licenses                                                                                                                                                                             |
| PCLP-3601 | Exceeding maximum <code>#include</code> depth is now an unsuppressible fatal error                                                                                                                                   |
| PCLP-3604 | Use of deprecated types diagnosed outside of declarations                                                                                                                                                            |
| PCLP-3605 | Support for MISRA C 2012 AMD-2                                                                                                                                                                                       |
| PCLP-3616 | Perform case-insensitive matching for <code>-efile/+efile</code> when <code>fff</code> flag option is ON                                                                                                             |
| PCLP-3629 | Lint comments and macros subject to AST suppressions from special forms of <code>-emacro</code> ( <code>(#)</code> or <code>#</code> ) within macro argument expansions are now processed at each argument expansion |
| PCLP-3636 | Improve rendering of references in message help                                                                                                                                                                      |
| PCLP-3639 | Improved compiler configuration for IAR ARM                                                                                                                                                                          |
| PCLP-3640 | Allow enclosing expression suppressions at the start of an expression                                                                                                                                                |
| PCLP-3641 | Update <code>pclp_config</code> support for Visual Studio 2019 16.7.1 to resolve error 5557 in the <code>xutility</code> header                                                                                      |
| PCLP-3642 | Performance improvements when using a large number of macros                                                                                                                                                         |
| PCLP-3647 | Clarify support for MISRA guidelines regarding assistance in identification of binary pointer operations                                                                                                             |
| PCLP-3655 | Remove reference to non-existent message                                                                                                                                                                             |
| PCLP-3663 | Improve wording of 4xxx and 5xxx messages section                                                                                                                                                                    |
| PCLP-3666 | Make instructions for suppressing coding guideline checking in library files more prominent                                                                                                                          |
| PCLP-3690 | Correct built-in semantics documentation                                                                                                                                                                             |
| PCLP-3691 | Added missing built-in semantics for Qt's <code>QMutex</code> and <code>QRecursiveMutex</code> member functions.                                                                                                     |
| PCLP-3693 | Improve value tracking of pointer casts from derived class to base class                                                                                                                                             |
| PCLP-3695 | Resolve false positives related to switch statements nested within loops                                                                                                                                             |
| PCLP-3698 | Resolve false positive 438 for value used within switch case that returns                                                                                                                                            |
| PCLP-3701 | Resolve unexpected interpretations of return values of conversion operators to integral types                                                                                                                        |
| PCLP-3703 | Document the <code>thread_atomic</code> , <code>thread_non_atomic</code> , and <code>thread_protected</code> semantics                                                                                               |
| PCLP-3705 | Correct if statement lock state inferencing                                                                                                                                                                          |
| PCLP-3706 | Resolve false positive 644 for variable initialized in all paths of switch statement except those that terminate the program                                                                                         |

|           |                                                                                                                                                   |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3712 | Include exception code in Windows crash message and resolve spurious pre-run crash messages caused by remote code injection by antivirus software |
| PCLP-3717 | Improve handling of function template instantiations for message 2467                                                                             |
| PCLP-3718 | Improve library classification of symbols with implicit declarations                                                                              |
| PCLP-3720 | Do not copy special mutex semantics when <code>-function</code> option copies function-wide semantics                                             |
| PCLP-3723 | Resolve false positive 530 on return values of functions with special mutex function semantic                                                     |
| PCLP-3732 | Improved intrinsic function support for XC compiler configurations generated with <code>pclp_config</code>                                        |
| PCLP-3733 | Correct the determination of const referenced parameters and atomic types                                                                         |
| PCLP-3738 | Resolve false positives involving union assignment within a loop and subsequent access to different subobject                                     |
| PCLP-3739 | Remove message 956 as a support mechanism for BARR-C:2018 Rule 1.8c                                                                               |
| PCLP-3741 | Resolve false positive 438 for variable assigned to prior to switch and read in switch case prior to fallthrough to next case                     |
| PCLP-3745 | Correct tracking of compound assignments for thread analysis                                                                                      |
| PCLP-3752 | Target macOS 10.15                                                                                                                                |
| PCLP-3753 | Resolve internal error involving incremented union members                                                                                        |
| PCLP-3754 | Add a section about potential interactions with antivirus software                                                                                |
| PCLP-3755 | Fix typo in Highlights section of 1.4 Beta 1 changes                                                                                              |
| PCLP-3757 | Remove references to non-existent message 9089 in <code>au-misra2.lnt</code> and <code>au-misra3.lnt</code>                                       |
| PCLP-3758 | Handle class member mutexes passed to locked class constructor                                                                                    |
| PCLP-3761 | Improve handling of Strong Types within one-way hierarchies joined using a binary operator and introduce <code>+ffj</code>                        |
| PCLP-3764 | Suppress spurious empty lines emitted in configurations generated with <code>pclp_config</code>                                                   |
| PCLP-3775 | Update Common Issues section and add FAQ section                                                                                                  |
| PCLP-3788 | Improve handling of exceptions to rules supported by message 970                                                                                  |
| PCLP-3797 | Parameterize messages 413 and 613 by symbol for member access expressions                                                                         |

### 23.5.1.3 Bugs Fixed

|           |                                                                                                                                                                                                                                     |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2086 | False negative 529 or 550 and false positive 953 for condition variables                                                                                                                                                            |
| PCLP-2097 | False positive 644 for structure member initialized by conditional structure assignment in C                                                                                                                                        |
| PCLP-2573 | Message 651 no longer issued for empty nested aggregate initialization                                                                                                                                                              |
| PCLP-2814 | Improvements to fatal error handling                                                                                                                                                                                                |
| PCLP-2826 | False positive 1576 for specialization via macro expansion                                                                                                                                                                          |
| PCLP-3189 | False negative 839                                                                                                                                                                                                                  |
| PCLP-3196 | Resolve an issue where a function that can indirectly call itself recursively where the recursive call occurs in the call chain below a recursive call to a different function could previously have been labeled "calls recursive" |
| PCLP-3221 | Resolve false positives associated with inferences regarding the right operand of a logical operator in a condition within a loop in the presence of knowledge of prior values                                                      |
| PCLP-3222 | Improper handling of symbols referenced through a using declaration                                                                                                                                                                 |
| PCLP-3305 | Message 1785 incorrectly issued for dependent types                                                                                                                                                                                 |
| PCLP-3391 | Improvements to determination of enclosing expressions for <code>--emacro((#),...)</code> suppressions                                                                                                                              |
| PCLP-3486 | Resolve false positive 929 for initialization of pointer to auto                                                                                                                                                                    |
| PCLP-3504 | <code>-libdir/+libdir</code> options not matching directories specified with a trailing slash                                                                                                                                       |
| PCLP-3519 | False positive 977 for call to overloaded operator with Boolean parameter                                                                                                                                                           |
| PCLP-3520 | False positive 586 for lambda with deduced return type                                                                                                                                                                              |
| PCLP-3593 | <code>nonnull</code> attribute incorrectly removes preexisting function argument semantics                                                                                                                                          |
| PCLP-3599 | False positive 586 for template argument substitution using <code>typedef</code>                                                                                                                                                    |
| PCLP-3613 | Messages inappropriately suppressed following fatal error                                                                                                                                                                           |

|           |                                                                                                                                                    |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3679 | Resolve an error in the handling of a semantic expression in specific situations involving a left operand of zero and a return semantic identifier |
| PCLP-3692 | Faulty interaction between <code>-u/--u</code> options and <code>-env_save/-env_restore/-env_push/-env_pop</code>                                  |
| PCLP-3694 | Message 9067 no longer issued for dependent-sized arrays or arrays with non-external linkage                                                       |
| PCLP-3696 | Resolve internal error or crash that could occur when analyzing invalid code containing syntax errors within a condition variable declaration      |
| PCLP-3725 | Restore parameterized suppression state when <code>-env_restore/-env_pop</code> options are used outside a module                                  |
| PCLP-3790 | Resolve false positive 8528 ( <code>au-barr.lnt</code> ) for anonymous enumerations outside of typedefs                                            |
| PCLP-3798 | Resolve internal error involving compound assignment as operand to comparison                                                                      |

#### 23.5.1.4 Known Issues

|           |                                                                                                                                                                                                            |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2135 | False negative 9003 for static variables                                                                                                                                                                   |
| PCLP-2391 | False positive instance of message 2434 for dereference of pointer lexically subsequent to but excluded from control flow after an earlier conditional deallocation within a loop                          |
| PCLP-2603 | Message 767 is disabled                                                                                                                                                                                    |
| PCLP-2614 | False positive instance of message 522 for call to const member function but has external side-effects                                                                                                     |
| PCLP-2999 | False negative instance of message 1536 involving address of private array element                                                                                                                         |
| PCLP-3004 | Message 1565 does not report the status of sub-objects within data members of class type                                                                                                                   |
| PCLP-3190 | False negative instance of message 9011 when loop exit point count is exceeded only with consideration of exit points within nested loops                                                                  |
| PCLP-3197 | False negative instance of message 9034 involving compound assignment                                                                                                                                      |
| PCLP-3213 | False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter                                                              |
| PCLP-3223 | The undocumented <code>+fil</code> flag is not implemented                                                                                                                                                 |
| PCLP-3242 | Conditional operator expression where both the second and third operand are of the same enumeration type not treated as resulting in a value of that enumeration type in C for the purposes of message 641 |
| PCLP-3652 | Crash when using recent versions of the Asio C++ Library / Boost.Asio                                                                                                                                      |
| PCLP-3660 | False positive instance of message 776 for addition between int and long of equal size                                                                                                                     |
| PCLP-3787 | Potential syntax error when using <code>std::get&lt;T&gt;(std::variant)</code> or <code>std::holds_alternative</code> from the Visual Studio 2019 standard library                                         |

#### 23.5.1.5 AUTOSAR Summary

|           |                                                                     |
|-----------|---------------------------------------------------------------------|
| PCLP-3599 | False positive 586 for template argument substitution using typedef |
| PCLP-3546 | Improved support for AUTOSAR Rule A18-0-3                           |
| PCLP-3520 | False positive 586 for lambda with deduced return type              |

#### 23.5.1.6 CERT C Summary

|           |                                                                   |
|-----------|-------------------------------------------------------------------|
| PCLP-2744 | Added support for CERT recommendation CON01-C                     |
| PCLP-3558 | Disassociate inappropriate messages from CERT C guideline MSC12-C |
| PCLP-2762 | Detect dangerous accesses between threads                         |
| PCLP-2757 | Detect inconsistent mutex locking order                           |
| PCLP-2753 | Detect dangerous accesses between threads                         |

**23.5.1.7 MISRA Summary**

|           |                                                                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3788 | Improve handling of exceptions to rules supported by message 970                                                                          |
| PCLP-3757 | Remove references to non-existent message 9089 in <code>au-misra2.lnt</code> and <code>au-misra3.lnt</code>                               |
| PCLP-3647 | Clarify support for MISRA guidelines regarding assistance in identification of binary pointer operations                                  |
| PCLP-3605 | Support for MISRA C 2012 AMD-2                                                                                                            |
| PCLP-3192 | Improvements to message 9020 (header file name with non-standard character sequence)                                                      |
| PCLP-3189 | False negative 839                                                                                                                        |
| PCLP-2891 | Member hiding messages are no longer used to support MISRA C++ Rule 2-10-2                                                                |
| PCLP-2589 | Message 9035 will now report any use of VLAs                                                                                              |
| PCLP-3197 | False negative instance of message 9034 involving compound assignment                                                                     |
| PCLP-3190 | False negative instance of message 9011 when loop exit point count is exceeded only with consideration of exit points within nested loops |

**23.6 Version 1.3.5****23.6.1 Summary****23.6.1.1 Improvements**

|           |                                                                                                                                                                                                |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-964  | Add thread information to internal error and crash messages                                                                                                                                    |
| PCLP-1757 | Recognize explicit initialization using default constructor and add flag to consider implicit initialization using an explicitly declared default constructor to be an explicit initialization |
| PCLP-1820 | Support message 506 and improve support for message 717                                                                                                                                        |
| PCLP-2298 | New support for AUTOSAR A5-0-2 and M5-0-14 guidelines, improved MISRA C++ support for Rules 5-0-13 and 5-0-14                                                                                  |
| PCLP-2973 | Support for AUTOSAR Rule A12-8-7                                                                                                                                                               |
| PCLP-2975 | Support for AUTOSAR Rule A13-2-3                                                                                                                                                               |
| PCLP-2996 | Stack usage information now appears in deterministic order                                                                                                                                     |
| PCLP-3085 | Update documentation regarding manual configuration without <code>pcpl_config</code>                                                                                                           |
| PCLP-3101 | Performance improvements for range-based suppressions                                                                                                                                          |
| PCLP-3117 | Suppress messages issued inside of explicitly defaulted function bodies                                                                                                                        |
| PCLP-3119 | Message 981 no longer issued for template instantiations                                                                                                                                       |
| PCLP-3146 | Updated example used in description of message 1544                                                                                                                                            |
| PCLP-3170 | Improvements to <code>const</code> variable exemption opt-in interpretation                                                                                                                    |
| PCLP-3171 | Update description of message 1919                                                                                                                                                             |
| PCLP-3172 | Update description of message 1930                                                                                                                                                             |
| PCLP-3184 | Parameterized suppression and <code>+paraminfo</code> support for supplemental messages 894 and 897                                                                                            |
| PCLP-3185 | Document that messages 893, 894, and 897 are immune to direct one-line suppressions                                                                                                            |
| PCLP-3188 | Remove listing for unused supplemental message 890                                                                                                                                             |
| PCLP-3193 | Improved handling of array arguments for message 857                                                                                                                                           |
| PCLP-3194 | Improvements to support for A5-2-2                                                                                                                                                             |
| PCLP-3199 | Clarify that message 9075 operates on objects, not functions                                                                                                                                   |
| PCLP-3202 | Message 829 is now issued for multiple inclusion of headers employing <code>#pragma once</code> and is now applicable to precompiled headers                                                   |
| PCLP-3203 | Correct typo in description of message 9134                                                                                                                                                    |
| PCLP-3206 | Improve support for reporting block scope function declarations                                                                                                                                |
| PCLP-3207 | Improve handling of static storage duration variables when calling functions with the <code>assert</code> semantic                                                                             |
| PCLP-3212 | Report implicit conversions occurring via member-init-list initialization.                                                                                                                     |
| PCLP-3214 | Improved support for MISRA C 2004 Rule 12.8                                                                                                                                                    |
| PCLP-3218 | Message 9019 more accurately reports declarations appearing before an <code>#include</code> directive                                                                                          |
| PCLP-3241 | Added new section describing rules related to processing of parameterized suppressions                                                                                                         |

|           |                                                                                                                                                                                                                     |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3258 | Remove non-existent message 2409 from product documentation                                                                                                                                                         |
| PCLP-3269 | Improved support for MISRA C 2012 Rule 11.2                                                                                                                                                                         |
| PCLP-3289 | Update description for <code>+fai</code>                                                                                                                                                                            |
| PCLP-3303 | Added new/missing guidelines for CERT C                                                                                                                                                                             |
| PCLP-3311 | Support non-variadic functions with <code>printf</code> and <code>scanf</code> semantics                                                                                                                            |
| PCLP-3320 | Suppress messages issued during expansion of <code>assert</code> macro                                                                                                                                              |
| PCLP-3341 | Flexible refinements to tracking of implementation-defined pointers converted from integral values                                                                                                                  |
| PCLP-3359 | Consider shift operators to be bitwise operators for the purpose of rule 5-0-21                                                                                                                                     |
| PCLP-3368 | Improve support for Visual Studio 2019 update 16.4                                                                                                                                                                  |
| PCLP-3374 | Improved support for IAR compiler keywords in generated compiler configurations                                                                                                                                     |
| PCLP-3381 | New message 2536 warns when invalid character encoding is detected in a string literal                                                                                                                              |
| PCLP-3382 | Add section documenting supported encodings of input files                                                                                                                                                          |
| PCLP-3384 | Convert relative paths in include options generated from <code>cl.exe /I</code> and <code>-I</code> options into absolute paths when producing a project configuration using the <code>pclpvscfg.exe</code> utility |
| PCLP-3386 | Add description and example for the <code>non_custodial</code> argument semantic                                                                                                                                    |
| PCLP-3389 | Add note regarding C++ language standard compiler options for Visual Studio when generating a Visual Studio compiler configuration                                                                                  |
| PCLP-3394 | Incorrect use of <code>+rw</code> and <code>-rw_asgn</code> more gracefully handled                                                                                                                                 |
| PCLP-3396 | Use application directory as default location when requesting selection of <code>pclp_config.py</code> in <code>pclpvscfg</code>                                                                                    |
| PCLP-3397 | Add default filenames to <code>pclpvscfg</code> dialogs                                                                                                                                                             |
| PCLP-3400 | Improve handling of bit-fields with qualified types                                                                                                                                                                 |
| PCLP-3406 | Enable use of <code>pclpvscfg</code> with 64-bit versions of Microsoft Visual Studio 2012, 2010, 2008, and 2005                                                                                                     |
| PCLP-3407 | Allow language standard options to be controlled by option environments                                                                                                                                             |
| PCLP-3411 | New behavior for messages referring to internal buffers during module processing                                                                                                                                    |
| PCLP-3423 | Improved compatible type handling for message 857                                                                                                                                                                   |
| PCLP-3433 | Add automated configuration support and documentation for MetaWare <code>ccac</code>                                                                                                                                |
| PCLP-3450 | Improve selection of <code>ctype.h</code> functions over macros                                                                                                                                                     |
| PCLP-3467 | Remove outdated limitation from description of <code>+emacro</code>                                                                                                                                                 |
| PCLP-3468 | Include <code>-efunc</code> and <code>+efunc</code> in "How Suppression Options are Applied" section.                                                                                                               |
| PCLP-3471 | Correct typo in an instance of message 2425                                                                                                                                                                         |
| PCLP-3480 | New support for AUTOSAR17 Rule A5-5-1                                                                                                                                                                               |
| PCLP-3481 | Support for AUTOSAR Rule A7-1-3                                                                                                                                                                                     |
| PCLP-3483 | Support for AUTOSAR Rule A13-5-1                                                                                                                                                                                    |
| PCLP-3495 | Exclude static local variables from stack usage                                                                                                                                                                     |
| PCLP-3502 | Update Open Source Declarations                                                                                                                                                                                     |
| PCLP-3506 | Add new <code>+fls</code> flag to control handling of static local variables after external calls                                                                                                                   |
| PCLP-3509 | Update descriptions of several messages                                                                                                                                                                             |
| PCLP-3513 | Inclusion of PC-lint Plus functional safety certificates and Best Practices document                                                                                                                                |
| PCLP-3552 | Value of static variable lost when assigning return value of an unwalked function                                                                                                                                   |
| PCLP-3555 | Treat zero-length arrays as incomplete arrays for Value Tracking                                                                                                                                                    |
| PCLP-3565 | Improvements to compiler configurations generated for GCC by <code>pclp_config</code>                                                                                                                               |

### 23.6.1.2 Bugs Fixed

|           |                                                                                                                                         |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2812 | False positive 9045 for union definition                                                                                                |
| PCLP-2885 | False positive 550 for variables captures by reference in a lambda                                                                      |
| PCLP-3093 | Resolve an issue that could manifest as an incomplete entry in the stack usage report                                                   |
| PCLP-3145 | Improve handling of messages 438 and 838 within loops                                                                                   |
| PCLP-3176 | Fix false negative for message 2427                                                                                                     |
| PCLP-3179 | Fix missing location when reporting a declaration of <code>main</code> with the <code>inline</code> or <code>constexpr</code> specifier |
| PCLP-3180 | Improve stack usage for <code>switch</code> statements                                                                                  |
| PCLP-3181 | Resolve an issue where union type punning could result in undesirable uninitialized access messages                                     |

|           |                                                                                                                                                                                                               |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-3183 | Missing message 2707                                                                                                                                                                                          |
| PCLP-3187 | False negative 451 and 967 for header with controlling macro but no standard include guard.                                                                                                                   |
| PCLP-3215 | Resolve false positive 473 for address-of array subscript                                                                                                                                                     |
| PCLP-3224 | Resolve an issue where members of structures within unions could incorrectly be reported as deallocated                                                                                                       |
| PCLP-3229 | Resolve a potential crash involving self-referential constructor initializers                                                                                                                                 |
| PCLP-3234 | Incorrect option syntax in IAR compiler configurations generated with <code>pclp_config</code>                                                                                                                |
| PCLP-3294 | Do not issue 1537 when returning a function pointer                                                                                                                                                           |
| PCLP-3310 | False positive 593 for custodial pointer always closed/freed in <code>in/else</code> statement                                                                                                                |
| PCLP-3324 | Resolve an internal error involving arrays of incomplete types                                                                                                                                                |
| PCLP-3358 | Resolve a missed potential Boolean inference for a ternary operator condition when processing a C module                                                                                                      |
| PCLP-3364 | Resolve false negatives involving interval relational-or-equal comparisons                                                                                                                                    |
| PCLP-3376 | Fix possible crash after diagnosing a conflict between a return value allocated by an expression produced by a macro expansion involving the built-in <code>__LINE__</code> macro and a return value semantic |
| PCLP-3378 | Fix missing declaration issues when naming declarations with identifiers that were subject to <code>-rw</code> options in the presence of <code>lint</code> comments                                          |
| PCLP-3380 | Message 725 incorrectly issued as a warning                                                                                                                                                                   |
| PCLP-3388 | Corrected processing of implied <code>this</code> arguments ('t') within <code>-function</code> option.                                                                                                       |
| PCLP-3392 | Disable line breaks in Keil $\mu$ Vision ARMCC configuration to resolve IDE navigation issues with long lines                                                                                                 |
| PCLP-3434 | Fix an internal error that could occur when a template applies the parameter pack size operator to a non-type template parameter pack of references                                                           |
| PCLP-3442 | False positive 542 when assigning negative values to bitfields                                                                                                                                                |
| PCLP-3465 | <code>+efreeze/++efreeze</code> incorrectly prevents parameterized message enabling options from targeting frozen messages                                                                                    |
| PCLP-3466 | Ignore duplicate parameterized suppression options involving partially frozen message sets                                                                                                                    |
| PCLP-3469 | Crash when processing user-defined function semantic employing invalid argument index                                                                                                                         |
| PCLP-3470 | False positive 542 for bitfields of unsigned enumeration type                                                                                                                                                 |
| PCLP-3472 | Fix invalid UNC path expansion when using <code>%ENCLOSING_DIRECTORY%</code> within a file located on a Windows network drive.                                                                                |
| PCLP-3507 | Fix false positive 438 during specific walk                                                                                                                                                                   |
| PCLP-3543 | Corrected typos in <code>au-autosar.lnt</code>                                                                                                                                                                |
| PCLP-3547 | Incorrect handling of friend declarations in class templates                                                                                                                                                  |
| PCLP-3561 | Removed inappropriate supporting message for MISRA C++ Rule 6-6-3                                                                                                                                             |
| PCLP-3570 | Message 3702 not issued when message 3402 suppressed                                                                                                                                                          |

### 23.6.1.3 Known Issues

|           |                                                                                                                                                                                                                                                                                         |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2135 | False negative 9003 for static variables                                                                                                                                                                                                                                                |
| PCLP-3196 | A function that can indirectly call itself recursively where the recursive call occurs in the call chain below a recursive call to a different function may be labeled "calls recursive"                                                                                                |
| PCLP-3213 | False positive implicit conversion messages when an explicit cast expression is used as a template argument for a non-type template parameter                                                                                                                                           |
| PCLP-3221 | False positive 413 for conditional evaluation of dereference of pointer inferred to be non-null on the right side of a logical AND expression in the condition of an <code>if</code> statement within a <code>for</code> statement prior to which the pointer had been known to be null |
| PCLP-3223 | The undocumented <code>+fil</code> flag is not implemented                                                                                                                                                                                                                              |
| PCLP-3242 | Conditional operator expression where both the second and third operand are of the same enumeration type not treated as resulting in a value of that enumeration type in C for the purposes of message 641                                                                              |
| PCLP-3260 | Message 440 is disabled                                                                                                                                                                                                                                                                 |

PCLP-3519 False positive 977 for call to overloaded operator with Boolean parameter

#### 23.6.1.4 AUTOSAR

PCLP-3483 Support for AUTOSAR Rule A13-5-1  
 PCLP-3481 Support for AUTOSAR Rule A7-1-3  
 PCLP-3480 New support for AUTOSAR17 Rule A5-5-1  
 PCLP-3206 Improve support for reporting block scope function declarations  
 PCLP-3194 Improvements to support for A5-2-2  
 PCLP-2975 Support for AUTOSAR Rule A13-2-3  
 PCLP-2973 Support for AUTOSAR Rule A12-8-7  
 PCLP-2298 New support for AUTOSAR A5-0-2 and M5-0-14 guidelines, improved MISRA C++ support for Rules 5-0-13 and 5-0-14

#### 23.6.1.5 MISRA C 2012

PCLP-3450 Improve selection of ctype.h functions over macros  
 PCLP-3269 Improved support for MISRA C 2012 Rule 11.2

#### 23.6.1.6 MISRA C 2004

PCLP-3214 Improved support for MISRA C 2004 Rule 12.8

#### 23.6.1.7 MISRA C++

PCLP-3359 Consider shift operators to be bitwise operators for the purpose of rule 5-0-21  
 PCLP-3206 Improve support for reporting block scope function declarations  
 PCLP-3170 Improvements to const variable exemption opt-in interpretation  
 PCLP-2298 New support for AUTOSAR A5-0-2 and M5-0-14 guidelines, improved MISRA C++ support for Rules 5-0-13 and 5-0-14

### 23.7 Version 1.3

#### 23.7.1 Summary

##### 23.7.1.1 Bugs Fixed

PCLP-1806 Resolve false positives related to initialization of members of returned class objects  
 PCLP-1835 Internal error A9BE308E (100829) related to use of definition options and precompiled headers  
 PCLP-1972 False positive 1762  
 PCLP-2036 Recognize when unconditional transfer of control by nested switch statement contributes to unreachability of statement subsequent to enclosing switch statement  
 PCLP-2227 Name mangling error with no category for C compound literal within C++ template argument resolved  
 PCLP-2459 False positive 981 for cast to `_Bool` in C  
 PCLP-2521 Respect redundant list initialization in C++17 mode  
 PCLP-2560 Warning levels supported in `-zero_err` and `+zero_err` options  
 PCLP-2598 False positive 1762 for call to non-const function in member expression  
 PCLP-2621 Don't report issue 522/714/729/759/765 for static local symbols  
 PCLP-2624 Message 967 not always issued when expected  
 PCLP-2626 Message 1790 should consider indirect base classes  
 PCLP-2649 False positive 527 for jump statements within control structures within switch statement bodies  
 PCLP-2663 False positive 9103 involving function templates



|           |                                                                                                                                                                                                 |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2668 | Message 882 issued as a warning instead of informational                                                                                                                                        |
| PCLP-2669 | Presentation issues with messages 682 and 882                                                                                                                                                   |
| PCLP-2670 | Missing conditional return inference when returning ternary expression                                                                                                                          |
| PCLP-2703 | Assertion failure or crash when certain headers are included with a precompiled header                                                                                                          |
| PCLP-2704 | Message 2454 doesn't work as expected with <code>-etype</code>                                                                                                                                  |
| PCLP-2726 | Corrected handling of function parameters for clash detection                                                                                                                                   |
| PCLP-2742 | Don't warn when calling <code>fflush</code> with null pointer                                                                                                                                   |
| PCLP-2772 | Hang or internal error triggered by nested use of <code>-header</code> , lint comments, <code>-indirect</code> , and <code>-d</code>                                                            |
| PCLP-2778 | False positive precision-related messages involving shift by non-constant                                                                                                                       |
| PCLP-2786 | Don't issue 715 for functions with skipped bodies                                                                                                                                               |
| PCLP-2793 | Don't complain about use of <code>regparm</code> attribute                                                                                                                                      |
| PCLP-2795 | Correct handling of <code>~</code> operator in semantics specifications                                                                                                                         |
| PCLP-2806 | False positive 1927 for classes containing anonymous unions                                                                                                                                     |
| PCLP-2818 | False positive 850 for rvalue reference to for index variable                                                                                                                                   |
| PCLP-2844 | Corrected imposter.c for old C compilers                                                                                                                                                        |
| PCLP-2869 | Buffered output streams not flushed before redirection on Windows                                                                                                                               |
| PCLP-2894 | Hang when instantiating member function of class template with dependent return value needed for deduction of <code>auto</code> return type in dependent caller in Microsoft compatibility mode |
| PCLP-2897 | <code>pclp_config</code> now properly handles options specifying lowercase macros                                                                                                               |
| PCLP-2900 | False positive 734 (loss of precision) for excessive precision of cast result                                                                                                                   |
| PCLP-2921 | The <code>fff</code> flag is now ON by default for Windows builds                                                                                                                               |
| PCLP-2925 | Incorrect <code>-append</code> option in <code>au-misra3.lnt</code>                                                                                                                             |
| PCLP-2959 | Message 1932 includes correct base class                                                                                                                                                        |
| PCLP-2986 | False positive 9106 for imaginary literal suffixes                                                                                                                                              |
| PCLP-2998 | Improve parsing of constant expressions whose evaluation involves side effects                                                                                                                  |
| PCLP-3016 | Move incompatible pointer type message from 2450 to 2449                                                                                                                                        |
| PCLP-3021 | Difference between base configurations for 32-bit and 64-bit versions of Visual Studio 2013 and 2015                                                                                            |

### 23.7.1.2 Documentation Improvements

|           |                                                                                  |
|-----------|----------------------------------------------------------------------------------|
| PCLP-2559 | Correct description of <code>-zero_err</code> and <code>+zero_err</code> options |
| PCLP-2561 | Document behavior of pattern consisting entirely of wild card characters         |
| PCLP-2583 | Document variants of message 686 for suspicious <code>-i</code> options          |
| PCLP-2634 | Updated description for 1564                                                     |
| PCLP-2641 | Improve documentation for <code>+fpe</code>                                      |
| PCLP-2804 | Update MISRA support tables                                                      |
| PCLP-2811 | Improve 701-704 message documentation                                            |
| PCLP-2822 | Messages 1526 and 1714 not issued for private members                            |
| PCLP-2825 | New MISRA Support Summaries                                                      |
| PCLP-2860 | Clarified meaning of <i>global</i> for messages 757 and 758.                     |
| PCLP-2864 | Clarify message description for 579                                              |
| PCLP-2883 | Diagnose suspicious uses of late multi-threaded output redirection options       |
| PCLP-2898 | Update and distinguish descriptions of 1764 and 1746                             |
| PCLP-2899 | Correct the citations for messages 901, 1731, and 1906.                          |
| PCLP-2905 | Update descriptions of <code>-save</code> and <code>-restore</code> options      |
| PCLP-2907 | Extend suspicious <code>-i</code> warnings to <code>--i</code>                   |
| PCLP-2911 | Document the fact that suppression options are not retained in PCH files.        |
| PCLP-2916 | Correct decidable classification for MISRA C 2012 Rule 11.2                      |
| PCLP-3003 | Update description of message 1774                                               |
| PCLP-3010 | Version and beta status added to the manual                                      |

### 23.7.1.3 General Improvements

|           |                                                                                                     |
|-----------|-----------------------------------------------------------------------------------------------------|
| PCLP-744  | Display current size options when printing help                                                     |
| PCLP-1344 | Allow <code>return</code> and <code>goto</code> to replace <code>break</code> in switches for MISRA |

|           |                                                                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-1351 | Message 9106 now includes full MS integer literal suffix                                                                                  |
| PCLP-1944 | Updates and clarifications to message 912                                                                                                 |
| PCLP-1973 | Improve handling of forward goto                                                                                                          |
| PCLP-2005 | Improvements to handling of list initialization for message 3701                                                                          |
| PCLP-2107 | Value Tracking for compound literals                                                                                                      |
| PCLP-2215 | Ignore framework directory suffix in clang search include path for <code>pclp_config</code> on macOS                                      |
| PCLP-2250 | Improve support for chained cleanup                                                                                                       |
| PCLP-2270 | Message 778 no longer issued inside of static assertions                                                                                  |
| PCLP-2415 | Extend inferencing to variables within arithmetic expressions                                                                             |
| PCLP-2418 | Improved support for IAR compilers                                                                                                        |
| PCLP-2461 | Reduce false positives in Value Tracking related to implications of if statement conditions within called functions during loop iteration |
| PCLP-2462 | Consider use of value through reference to different type                                                                                 |
| PCLP-2470 | Report 445 (re-use of for index variable) for arbitrary levels of nesting                                                                 |
| PCLP-2485 | Add optional string parameter to message 9022                                                                                             |
| PCLP-2511 | Improved handling of references to structures for message 1764                                                                            |
| PCLP-2536 | Improved Value Tracking for unions                                                                                                        |
| PCLP-2568 | <code>pclp_config</code> extracts values of Microsoft C/C++ compiler macros                                                               |
| PCLP-2586 | Reduced false positives of "not used" messages involving variables read and written during loops                                          |
| PCLP-2587 | Improve calculation of unsigned addition of inferred values with indeterminate overflow                                                   |
| PCLP-2597 | New variable/function/macro parameter for message 774                                                                                     |
| PCLP-2607 | Refinements to template handling for message 2701                                                                                         |
| PCLP-2610 | Messages 548 and 9013 no longer issued in function template instantiations                                                                |
| PCLP-2622 | Messages 759 and 765 now issued for single modules                                                                                        |
| PCLP-2627 | Message 9098 softened for casts                                                                                                           |
| PCLP-2628 | Enhancements to messages 907 and 908                                                                                                      |
| PCLP-2631 | Improved error message for unknown compiler name for <code>pclp_config</code>                                                             |
| PCLP-2632 | Expand scope of messages 449 and 2434                                                                                                     |
| PCLP-2635 | Clarify text of message 9079                                                                                                              |
| PCLP-2638 | Harmonize text for variants of message 648                                                                                                |
| PCLP-2639 | Fix missing "C++" text in message descriptions for <code>-help</code> and <code>-dump_messages</code> options.                            |
| PCLP-2658 | Message 9233 strengthened for shifts by a known negative value                                                                            |
| PCLP-2659 | Message 981 is not issued for conversions to class type                                                                                   |
| PCLP-2664 | Refinement to message 1762 for closure members                                                                                            |
| PCLP-2672 | Issues message 305 when failing to open a file                                                                                            |
| PCLP-2697 | Improved message text for <code>-dump_messages</code> and <code>-dump_message_list</code>                                                 |
| PCLP-2698 | Support for the <code>__has_unique_object_representations</code> type trait intrinsic                                                     |
| PCLP-2729 | Support <code>_Float128</code> for GCC configurations generated with <code>pclp_config</code> .                                           |
| PCLP-2773 | Message 866 no longer issued for dependent expressions                                                                                    |
| PCLP-2784 | Allow reuse of indirect files in different option environments                                                                            |
| PCLP-2803 | References to non-existent messages removed from MISRA author files                                                                       |
| PCLP-2805 | Added reference to MISRA C 2012 AMD-1 Directive 4.14 to <code>au-misra3-amd1.lnt</code>                                                   |
| PCLP-2809 | Sort the 'could be const' messages by source location                                                                                     |
| PCLP-2813 | SGML terminating tags are now written when exiting from a fatal error                                                                     |
| PCLP-2821 | Warn about call through null function pointer                                                                                             |
| PCLP-2823 | Messages 1931 and 9169 no longer issued for deleted constructors                                                                          |
| PCLP-2833 | Improved exception handling diagnostics                                                                                                   |
| PCLP-2835 | Add space in text of message 1705                                                                                                         |
| PCLP-2839 | False negative 1762 due to calling static member function without visible definition                                                      |
| PCLP-2846 | Extend situations where message 464 is reported                                                                                           |
| PCLP-2848 | Suppress 1746 for small trivially-copyable types                                                                                          |
| PCLP-2868 | Warn about default constructors defaulted outside their class                                                                             |
| PCLP-2875 | Exempt friend declarations from message 1784                                                                                              |
| PCLP-2886 | Improved analysis of 'new' expressions for 1556 and 1764                                                                                  |
| PCLP-2889 | Improved 'pclp_config' support for older versions of Visual Studio                                                                        |
| PCLP-2896 | Version check in MISRA configuration files                                                                                                |
| PCLP-2906 | Diagnostic when <code>-restore</code> is used without <code>-save</code>                                                                  |

|           |                                                                                                                                                     |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2927 | Extend implicit assignment context messages to in-class non-static data member initializers                                                         |
| PCLP-2928 | Support <code>-isystem</code> for GCC and clang compilers.                                                                                          |
| PCLP-2937 | Multiple improvements for handling of warning levels.                                                                                               |
| PCLP-2940 | Ignore single redundant unreachable break at end of switch case for message 527 by default                                                          |
| PCLP-2942 | Message 9168 no longer reported for declarations with incomplete types                                                                              |
| PCLP-2960 | Message 1932 no longer issued for forward declarations or template instantiations                                                                   |
| PCLP-2962 | Reduced memory usage                                                                                                                                |
| PCLP-2977 | Multiple improvements for message 1915                                                                                                              |
| PCLP-2984 | Improved performance and reduced memory usage when using many <code>-append</code> and <code>-deprecate</code> options or MISRA configuration files |
| PCLP-2989 | Improved performance                                                                                                                                |
| PCLP-3002 | Update default C++ standard from C++14 ( <code>-std=c++14</code> ) to C++17 ( <code>-std=c++17</code> )                                             |
| PCLP-3014 | Issue 1774 for comparison via equality operators                                                                                                    |
| PCLP-3042 | Improvements to <code>+fsf</code> flag and semantic monikers for constructors and destructors of class templates                                    |

#### 23.7.1.4 MISRA C 2012 Improvements

|           |                                                                                                                        |
|-----------|------------------------------------------------------------------------------------------------------------------------|
| PCLP-2060 | Support MISRA C 2012 AMD-1 Rule 21.18                                                                                  |
| PCLP-2537 | Treat members of different tagless structures as appearing in separate name spaces for 9046                            |
| PCLP-2777 | Message 931 removed from MISRA C 2012 rule 13.2                                                                        |
| PCLP-2791 | Improved support for MISRA C 2012 Rules 17.1, 21.4, 21.5, and 21.10                                                    |
| PCLP-2797 | Support MISRA C 2012 Rules 22.4 and 22.6                                                                               |
| PCLP-2807 | Message 774 added to MISRA C 2012 rule 2.2                                                                             |
| PCLP-2914 | Improved support for MISRA C 2012 Rule 11.1                                                                            |
| PCLP-2915 | Improved support for MISRA C 2012 Rule 22.1                                                                            |
| PCLP-2918 | Improved support for MISRA C 2012 Rule 14.4                                                                            |
| PCLP-2919 | Improved support for MISRA C 2012 Rule 15.7                                                                            |
| PCLP-2920 | Consider Boolean strong type hierarchy beyond the Boolean strong type itself for Boolean essential type classification |
| PCLP-2926 | Improved handling of Boolean MISRA C 2012 essential type for <code>false</code> and <code>true</code> macros           |

#### 23.7.1.5 MISRA C++ Improvements

|           |                                                                                                                                              |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2700 | Message 9114 (MISRA C++ 5-0-3) now reports violations of the rule even when only the underlying type is converted and the actual type is not |
| PCLP-2705 | Improved support for MISRA C++ Rule 7-1-1                                                                                                    |
| PCLP-2741 | Improved support for MISRA C++ Rule 5-2-4                                                                                                    |
| PCLP-2774 | Correct false positive 1536 messages                                                                                                         |
| PCLP-2798 | Improved support for compound assignment operators for rules 5-0-3 and 5-0-6                                                                 |
| PCLP-2832 | Improved support for MISRA C++ Rule 3-9-3                                                                                                    |
| PCLP-2877 | Support digit-related exceptions for character arithmetic                                                                                    |
| PCLP-2880 | Ignore all extern "C" declarations for message 9141 (7-3-1)                                                                                  |
| PCLP-2881 | New interpretation option to restrict definition of constant expression for the purposes of underlying type determination                    |
| PCLP-2892 | Only report casts in one direction for MISRA C++ 2008 rule 5-2-8                                                                             |
| PCLP-2908 | Improved support for MISRA C++ Rule 7-1-2                                                                                                    |
| PCLP-2939 | Improved support for MISRA C++ 2008 Rule 4-10-2                                                                                              |
| PCLP-2943 | Improved support for MISRA C++ Rule 2-13-2                                                                                                   |

#### 23.7.1.6 New Features

|           |                                                                                                                                     |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2022 | Add <b>+ffi</b> to format large integer values and ranges relative to nearby integer limits                                         |
| PCLP-2173 | Introductory support for CERT C                                                                                                     |
| PCLP-2193 | Track values of objects of pointer-to-member type in Value Tracking                                                                 |
| PCLP-2247 | Add <b>+fcv</b> to exempt parameters whose only reference was a cast to void from messages indicating that they could be made const |
| PCLP-2333 | Add options to accommodate relative paths in the <b>imposter</b> utility                                                            |
| PCLP-2477 | Support for <b>0.NaN</b> and <b>0.Infinity</b>                                                                                      |
| PCLP-2483 | New message 2666 - expression with side effects passed to unexpanded parameter of macro                                             |
| PCLP-2484 | New message 9501 - preprocessing directive in call to function                                                                      |
| PCLP-2488 | New message 973 - unary operator in macro not parenthesized                                                                         |
| PCLP-2503 | New message 823 - definition of macro ends in semi-colon                                                                            |
| PCLP-2562 | New message 9040 - possible struct hack declaration for member                                                                      |
| PCLP-2571 | New message 9502 - multi-statement macro is not enclosed in monocarpic do-while loop                                                |
| PCLP-2577 | New message 3419 - in-class initializer for static data member of floating type is a GNU extension                                  |
| PCLP-2625 | Support for BARR-C:2018 (previously Netrino) coding guidelines                                                                      |
| PCLP-2699 | New semantics and messages for signals                                                                                              |
| PCLP-2713 | New random number generator and seed function messages with new noliteral argument semantic                                         |
| PCLP-2739 | Built-in function semantics for dangerous Microsoft Windows API IsBadPtr functions                                                  |
| PCLP-2789 | New <b>non_custodial</b> semantic                                                                                                   |
| PCLP-2830 | New <b>+fup</b> and <b>+fuu</b> flags to clear values after reporting them as null or uninitialized                                 |
| PCLP-2836 | New message 695 - inline function declared without storage-class specifier                                                          |
| PCLP-2888 | Visual Studio 2019 support for <b>pclp_config</b>                                                                                   |
| PCLP-2910 | New message 1423 - <b>reinterpret_cast</b> has undefined behavior                                                                   |
| PCLP-2929 | New AUTOSAR C++ configuration file                                                                                                  |
| PCLP-2948 | New message 1756 - variable has static/thread storage duration and non-POD type                                                     |
| PCLP-2949 | New message 9215 and added parameter to message 715                                                                                 |
| PCLP-2950 | New message 9414 - 'typeid' used on expression with side effect                                                                     |
| PCLP-2951 | New message 9181 - switch contains fewer than two non-default switch cases                                                          |
| PCLP-2952 | New message 2618 - non-type specifier appears after a type                                                                          |
| PCLP-2954 | New message 9418 - enum does not have an explicitly specified underlying type                                                       |
| PCLP-2955 | New message 9419 - enum is not a scoped enumeration                                                                                 |
| PCLP-2956 | New message 9415 - 'auto' variable initialized using list initialization                                                            |
| PCLP-2957 | New message 9420 - bitfield does not have unsigned integer or explicitly unsigned enumeration type                                  |
| PCLP-2958 | New message 9432 - class inherits from multiple non-abstract classes                                                                |
| PCLP-2963 | Support deprecation of preprocessor directives                                                                                      |
| PCLP-2966 | New message 9422 - virtual function should specify exactly one of 'virtual', 'override', or 'final'                                 |
| PCLP-2967 | New message 9421 - virtual function overrides function and is not marked with 'override' or 'final'                                 |
| PCLP-2968 | New message 1779 - virtual function introduced in class which is marked as 'final'                                                  |
| PCLP-2970 | New message 9437 - non-POD class defined with 'struct' keyword                                                                      |
| PCLP-2971 | New message 9435 - symbol declared as friend                                                                                        |
| PCLP-2974 | New messages to diagnose literal operators and their use                                                                            |
| PCLP-2976 | New message 9436 - symbol declared with array type in C++ module                                                                    |
| PCLP-2983 | New message 2414                                                                                                                    |
| PCLP-3015 | Visual Studio configuration GUI for <b>pclp_config</b>                                                                              |
| PCLP-3017 | <b>pclp_config</b> support for Microchip MPLAB X                                                                                    |
| PCLP-3018 | <b>pclp_config</b> support for Keil µVision ARMCC                                                                                   |
| PCLP-3019 | New <b>%ENCLOSING_DIRECTORY%</b> built-in environment variable                                                                      |
| PCLP-3044 | New CERT C configuration file                                                                                                       |

## 23.8 Version 1.2

### 23.8.1 Summary

#### 23.8.1.1 Bugs Fixed

|           |                                                                                                       |
|-----------|-------------------------------------------------------------------------------------------------------|
| PCLP-1904 | False positive 9007                                                                                   |
| PCLP-1957 | False positive 915/917 for enumeration constants in C mode                                            |
| PCLP-1997 | False positive 9168 for static array member defined outside of class                                  |
| PCLP-2026 | Non-library regions incorrectly treated as library                                                    |
| PCLP-2099 | Constrain ranges of inferenced values in Value Tracking messages                                      |
| PCLP-2120 | Support the 'l' strong type flag                                                                      |
| PCLP-2158 | False positive 1539 for template                                                                      |
| PCLP-2183 | Improved recognition of conditional variable modification for Value Tracking                          |
| PCLP-2190 | Respect softeners for pointer differences when assigning or joining strong types                      |
| PCLP-2192 | Result of bitwise OR not always calculated correctly                                                  |
| PCLP-2196 | False positive 5731 for <code>__thread</code> in template                                             |
| PCLP-2219 | Improvements to handling of list initialization                                                       |
| PCLP-2226 | False positive 527 for conditional break in switch case                                               |
| PCLP-2229 | False positive 438 for variable used in implicit construction of <code>std::initializer_list</code>   |
| PCLP-2238 | False positive 733 when assigning to pointer parameter in specific walk                               |
| PCLP-2241 | False positive 9042 and 9082 for switch statements with leading <code>default</code> case             |
| PCLP-2246 | Message 967 not suppressed for <code>-header</code> files using library suppressions                  |
| PCLP-2249 | Errors issued for uninstantiated templates                                                            |
| PCLP-2258 | False positive 958 for structure with union members                                                   |
| PCLP-2261 | False negative 737 for equality/relational tests                                                      |
| PCLP-2262 | False positive 9090 when throwing a temporary object                                                  |
| PCLP-2268 | False positive 587 for expressions with bitwise operations on types with different size or signedness |
| PCLP-2271 | False positive 2434 for specific circumstances involving delete and return                            |
| PCLP-2278 | Change 1997 to 1998 in set of permitted C++ years for <code>-std</code>                               |
| PCLP-2279 | False positive 1529 for potentially unresolved overloaded operator&                                   |
| PCLP-2289 | Message 491 incorrectly issued for invalid name in macro definition                                   |
| PCLP-2315 | Crash when using <code>-h2</code> with empty caret indicator                                          |
| PCLP-2366 | Improved recognition of side effects from <code>std::initializer_list</code> initialization           |
| PCLP-2400 | Message 611 no longer issued for implicit conversions                                                 |
| PCLP-2417 | Only emit "Resuming file" verbosity messages with <code>-v&lt;integer&gt;</code>                      |
| PCLP-2431 | Correct handling of <code>-efunc</code> and <code>+efunc</code> options                               |
| PCLP-2445 | False positive 641 for parenthesized enumeration constant in C mode                                   |
| PCLP-2447 | False positive 743 (negative character constant) in template instantiation                            |

#### 23.8.1.2 MISRA C 2012 Improvements

|           |                                             |
|-----------|---------------------------------------------|
| PCLP-1059 | Support for MISRA C 2012 Rule 20.12         |
| PCLP-1061 | Support for MISRA C 2012 Rule 9.2           |
| PCLP-1066 | Support for MISRA C 2012 Rule 17.5          |
| PCLP-2176 | Improved support for MISRA C 2012 Rule 11.7 |
| PCLP-2443 | Improved support for MISRA C 2012 Rule 11.9 |
| PCLP-2455 | Support for MISRA C 2012 Rule 5.1           |
| PCLP-2456 | Support for MISRA C 2012 Rule 5.2           |
| PCLP-2457 | Support for MISRA C 2012 Rule 5.4           |

#### 23.8.1.3 MISRA C++ Improvements

|           |                                                                               |
|-----------|-------------------------------------------------------------------------------|
| PCLP-545  | Added support for MISRA C++ Rule 15-3-6                                       |
| PCLP-1953 | Implement MISRA's amended wording for balancing binary operators in MISRA C++ |
| PCLP-2263 | Improved support for MISRA C++ Rule 6-4-3                                     |
| PCLP-2266 | Improved support for MISRA C++ Rule 6-4-6                                     |
| PCLP-2454 | Void pointers no longer reported for Rule 5-2-7                               |

### 23.8.1.4 General Improvements

|           |                                                                                                                                                                                                                              |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PCLP-41   | Support for 16-bit and 32-bit bytes                                                                                                                                                                                          |
| PCLP-1659 | Extended exemptions for message 785                                                                                                                                                                                          |
| PCLP-1761 | Consider base class fields for message 1401                                                                                                                                                                                  |
| PCLP-1875 | Error 4374 suppressed for Visual Studio configurations                                                                                                                                                                       |
| PCLP-1885 | Tracking of multiple initialization variables in a <code>for</code> statement                                                                                                                                                |
| PCLP-1887 | Message 571 no longer issued for enumeration types                                                                                                                                                                           |
| PCLP-1895 | Soften 1938 for static-local and const-initialized variables                                                                                                                                                                 |
| PCLP-1948 | Better recognition of lint comments                                                                                                                                                                                          |
| PCLP-2119 | Extend the strong type 'z' softener to casts of null pointer constants                                                                                                                                                       |
| PCLP-2133 | Honor the value of the <code>fcc</code> flag option for summary output                                                                                                                                                       |
| PCLP-2147 | Improved Value Tracking inferencing for booleans                                                                                                                                                                             |
| PCLP-2160 | <code>-fiz</code> no longer affects initialization of booleans                                                                                                                                                               |
| PCLP-2180 | Issue 716 and not 774 for <code>while (1)</code> and <code>while (true)</code>                                                                                                                                               |
| PCLP-2181 | Only issue 1768 once per function                                                                                                                                                                                            |
| PCLP-2182 | Extend value tracking depth for <code>constexpr</code> functions                                                                                                                                                             |
| PCLP-2189 | Improved diagnostics for misuse of <code>-a</code> and <code>-s</code> options                                                                                                                                               |
| PCLP-2218 | Suppress message 948 for <code>if constexpr</code> conditions                                                                                                                                                                |
| PCLP-2233 | Don't issue 587, 685, or 837 in instantiations                                                                                                                                                                               |
| PCLP-2237 | New warnings for improper use of <code>-i</code>                                                                                                                                                                             |
| PCLP-2252 | Support <code>-d/-u</code> options within files included via <code>-indirect</code> and improve behavior of combining <code>-env_push/-env_pop</code> , <code>-env_save/-env_restore</code> , and <code>-d/-u</code> options |
| PCLP-2253 | Supplemental messages for compiler errors                                                                                                                                                                                    |
| PCLP-2254 | Improved diagnostics for misuse of <code>-strong</code> boolean options                                                                                                                                                      |
| PCLP-2260 | Unhelpful 746 when calling built-in atomic intrinsics in dependent contexts                                                                                                                                                  |
| PCLP-2265 | Only issue message 9139 once per switch                                                                                                                                                                                      |
| PCLP-2273 | Documentation improvements for VS2017 <code>pclp_config</code> configuration                                                                                                                                                 |
| PCLP-2302 | Added symbol parameter to 9018                                                                                                                                                                                               |
| PCLP-2304 | Consider tags used in <code>__builtin_offsetof</code> to be referenced                                                                                                                                                       |
| PCLP-2307 | Support testing for non-null before deleting pointer                                                                                                                                                                         |
| PCLP-2311 | Improved validation of the <code>+group</code> option                                                                                                                                                                        |
| PCLP-2312 | Improved location information for message 9049                                                                                                                                                                               |
| PCLP-2328 | Improved error handling for <code>pclp_config</code>                                                                                                                                                                         |
| PCLP-2339 | Add operator argument to message 514                                                                                                                                                                                         |
| PCLP-2346 | Message 857 softened for casts                                                                                                                                                                                               |
| PCLP-2388 | Recognize <code>std::addressof</code> for 1529                                                                                                                                                                               |
| PCLP-2398 | Suppressing 893 with <code>-estring</code>                                                                                                                                                                                   |
| PCLP-2401 | Suppress message 1506 in <code>final</code> classes                                                                                                                                                                          |
| PCLP-2412 | Improved value tracking inferencing                                                                                                                                                                                          |
| PCLP-2413 | Name shadowing involving enumeration constants now reported by 578                                                                                                                                                           |
| PCLP-2416 | Increased scope of message 445                                                                                                                                                                                               |
| PCLP-2420 | False positive 9107 for member function template instantiated in a module                                                                                                                                                    |
| PCLP-2438 | Message 1773 now issued for references                                                                                                                                                                                       |
| PCLP-2446 | Issue 9045 messages in a deterministic order                                                                                                                                                                                 |
| PCLP-2469 | Message 750 no longer reported when used in short-circuited <code>defined</code> operator                                                                                                                                    |
| PCLP-2478 | Support for response files introduced by compiler-specific option                                                                                                                                                            |

### 23.8.1.5 New Features

|           |                                                                                                            |
|-----------|------------------------------------------------------------------------------------------------------------|
| PCLP-1052 | Added <code>-idlen</code> and message 621                                                                  |
| PCLP-1128 | New Precision and Pre-determined Predicate Implementations                                                 |
| PCLP-1853 | Message 1757 added                                                                                         |
| PCLP-1994 | New <code>-misra_interpret</code> option to apply alternative interpretations to MISRA Rules               |
| PCLP-2177 | New <code>fcw</code> flag option to control whether 438 considers writes from called functions             |
| PCLP-2239 | New diagnostic for tentative array definition without a size in C mode                                     |
| PCLP-2267 | Improved C++17 Support                                                                                     |
| PCLP-2275 | New <code>fgl</code> flag option to control the use of GNU line markers in preprocessed output             |
| PCLP-2280 | New <code>fmt</code> flag option to enable matching of template template-arguments to compatible templates |

|           |                                                                                                     |
|-----------|-----------------------------------------------------------------------------------------------------|
| PCLP-2283 | New <code>fmx</code> flag allows disabling of C++ member access control                             |
| PCLP-2284 | New <code>fzd</code> flag to enable C++14 sized deallocation                                        |
| PCLP-2357 | New <code>-format_category</code> option to configure category representation                       |
| PCLP-2387 | Add <code>\e</code> escape sequence for inserting ASCII escape into format strings                  |
| PCLP-2428 | New message 2662 reports out of bounds pointer from scalar pointer                                  |
| PCLP-2508 | New message 2650 reports when a constant is out of range for part of a compound comparison operator |

### 23.8.1.6 Documentation Enhancements

|           |                                                                                                 |
|-----------|-------------------------------------------------------------------------------------------------|
| PCLP-2272 | Update entry for messages 413 and 613 to include symbol                                         |
| PCLP-2282 | Better explain how lint comments in macro definitions are handled                               |
| PCLP-2301 | Add section about using backslash escapes in options                                            |
| PCLP-2396 | Add version information to option and flag option tables                                        |
| PCLP-2429 | Add section documenting minimum and recommended OS/hardware requirements                        |
| PCLP-2430 | Add section describing how message suppression options are applied                              |
| PCLP-2439 | Correct notes about <code>+emacro</code> , <code>+elibmacro</code> , and <code>+elibsymb</code> |
| PCLP-2444 | Add note about new <code>-efile</code> behavior to "What's new" section                         |

## 23.9 Version 1.1

### 23.9.1 Summary

#### 23.9.1.1 Bugs Fixed

|           |                                                                                                              |
|-----------|--------------------------------------------------------------------------------------------------------------|
| PCLP-1454 | Improved handling of <code>goto</code> for read/write analysis                                               |
| PCLP-1483 | Single line suppressions not always honored when location is macro name                                      |
| PCLP-1512 | Improved handling of <code>-emacro</code>                                                                    |
| PCLP-1711 | Don't create null inferences when comparing to the <code>this</code> pointer                                 |
| PCLP-1826 | False positive 9107 for member instantiations                                                                |
| PCLP-1841 | Message 923 not issued for <code>reinterpret_cast</code>                                                     |
| PCLP-1894 | False positive 641 for parenthesized enumeration constant in C mode                                          |
| PCLP-1908 | False positive 838 for multiple early returns                                                                |
| PCLP-1919 | Improved recognition of volatile assignment as an impurity                                                   |
| PCLP-1909 | False positive 705 for <code>%jd</code> conversion specifier with <code>intmax_t</code> defined as long long |
| PCLP-1910 | Statements without side effects not diagnosed in specific circumstances                                      |
| PCLP-1914 | Message 826 no longer issued for <code>dynamic_cast</code>                                                   |
| PCLP-1920 | Improved custodial semantics for reference parameters                                                        |
| PCLP-1921 | Custodial semantic not properly applied for operator call expressions                                        |
| PCLP-1942 | False positive 568/775 for reference member variables                                                        |
| PCLP-1943 | False positive 568 for reference types                                                                       |
| PCLP-1945 | Message 916 now only reports pointer-to-pointer conversions                                                  |
| PCLP-1955 | Improperly suppressed messages from library headers                                                          |
| PCLP-1960 | Messages 900 and 870 not issued when using 2 passes                                                          |
| PCLP-1974 | Support for bidirectional pre-loop inference test direction                                                  |
| PCLP-1993 | False positive 1924 for substituted non-type template parameters of enumeration type                         |
| PCLP-1995 | Reset position indicator character when using <code>-h2</code>                                               |
| PCLP-2001 | Improved handling of pointer parameters for message 733                                                      |
| PCLP-2013 | Message 2702 now issued regardless of suppression state of 528                                               |
| PCLP-2014 | False positive 449                                                                                           |
| PCLP-2015 | False positive 9048 for use of enum constant non-type template parameter                                     |
| PCLP-2029 | Improved handling of angle brackets in message 773                                                           |
| PCLP-2068 | Extraneous supplemental message                                                                              |
| PCLP-2069 | Don't analyze assembly statements                                                                            |
| PCLP-2073 | Crash when using allocation semantics in user-defined function semantics                                     |
| PCLP-2077 | Incorrect null inference for equality check against non-null pointer                                         |
| PCLP-2078 | Consistent diagnosis of null pointers for array indexing and pointer arithmetic                              |
| PCLP-2101 | False positive 9176 for implicit conversion of this pointer to base class pointer                            |

|           |                                                                                                                                  |
|-----------|----------------------------------------------------------------------------------------------------------------------------------|
| PCLP-2102 | False Positive 743 for wide character constants                                                                                  |
| PCLP-2105 | False positive 527 after switch containing conditional return                                                                    |
| PCLP-2106 | Improved Value Tracking of structures in C                                                                                       |
| PCLP-2108 | False positive 1751 for macro defined in header that expands to anonymous namespace in main source file                          |
| PCLP-2125 | Remove message 504 from MISRA author files                                                                                       |
| PCLP-2127 | False negative 1506                                                                                                              |
| PCLP-2128 | Add POD semantics to Standard C library functions                                                                                |
| PCLP-2140 | False positive 1415 for pointers to void                                                                                         |
| PCLP-2151 | False positive type alias differences for function template specializations                                                      |
| PCLP-2153 | Internal error for improper user-defined function return allocation semantics                                                    |
| PCLP-2155 | False positive 9034 involving compound assignment                                                                                |
| PCLP-2156 | Message 9003 no longer issued for local static variables                                                                         |
| PCLP-2157 | False positive 909 and 910 for certain casts                                                                                     |
| PCLP-2159 | False positive 1727 for member function of class template specializations                                                        |
| PCLP-2163 | Strong type of enumerator not recognized when typedef uses incomplete type                                                       |
| PCLP-2169 | False positive 758 for implicit instantiations                                                                                   |
| PCLP-2179 | Internal error related to message 9027 when issued in C++ code                                                                   |
| PCLP-2191 | Structure member initialization status not properly merged after being initialized in both branches of an if statement in C mode |
| PCLP-2203 | Improved Value Tracking of unknown function parameters of structure type                                                         |
| PCLP-2208 | Message 1415 issued for type-dependent arguments                                                                                 |

#### 23.9.1.2 MISRA C 2004 Improvements

|           |                                                                           |
|-----------|---------------------------------------------------------------------------|
| PCLP-1874 | Improved support for MISRA C 2004 Rule 13.1                               |
| PCLP-1882 | Improved support for MISRA C 2004 Rules 11.1 and 11.2                     |
| PCLP-1929 | Improved handling of MISRA C 2004 Rule 10.1                               |
| PCLP-2009 | Improved support for MISRA C 2004 Rule 10.1                               |
| PCLP-2018 | Improved support for MISRA C 2004 Rules 6.1 and 6.2                       |
| PCLP-2037 | Improved support for MISRA C 2004 Rules 6.1/6.2 and MISRA C++ Rule 5-0-11 |

#### 23.9.1.3 MISRA C 2012 Improvements

|           |                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------|
| PCLP-1037 | Improved support for MISRA C 2012 Rule 20.8                                                   |
| PCLP-1528 | Update messages used for Rule 11.7 in au-misra3.lnt                                           |
| PCLP-1776 | Improved support for MISRA C 2012 Rule 10.3                                                   |
| PCLP-1778 | Improved support for MISRA C 2012 Rule 11.4                                                   |
| PCLP-1779 | Improved support for MISRA C 2012 Rule 11.4                                                   |
| PCLP-1780 | Improved support for MISRA C 2012 Rule 11.4                                                   |
| PCLP-1781 | Improved support for MISRA C 2012 Rule 11.5                                                   |
| PCLP-1879 | Improved support for MISRA C 2012 Rule 10.1                                                   |
| PCLP-1880 | Improved support for MISRA C 2012 Rule 10.1                                                   |
| PCLP-1881 | Improved support for MISRA C 2012 Rule 10.2                                                   |
| PCLP-1898 | Improved support for MISRA C 2012 Rule 10.7                                                   |
| PCLP-2017 | Do not classify wide character constants as having essentially character type in MISRA C 2012 |
| PCLP-2030 | Improved support for MISRA C 2012 Rule 18.8                                                   |
| PCLP-2198 | Incorrect MISRA essential type calculation for modulo operations                              |
| PCLP-2220 | Improved support for MISRA C 2012 Rule 11.5                                                   |

#### 23.9.1.4 MISRA C++ Improvements

|           |                                                                       |
|-----------|-----------------------------------------------------------------------|
| PCLP-1952 | Improved support for MISRA C++ Rule 5-3-4                             |
| PCLP-1989 | Improved support for MISRA C++ Rule 3-1-1                             |
| PCLP-2031 | Corrections to au-misra-cpp.lnt for MISRA C++ Rules 17-0-1 and 17-0-2 |
| PCLP-2117 | Improved support for MISRA C++ Rule 7-3-6                             |
| PCLP-2150 | False positive 9113 for compound assignment                           |



PCLP-2170 Increased scope of "related types" for message 9176

### 23.9.1.5 General Improvements

PCLP-1768 Improved handling when all paths return in `if` statement  
 PCLP-1790 Improved handling of placement `new` for read-write analysis  
 PCLP-1795 Improved handling of mutable members in side effects determination  
 PCLP-1799 Improved handling of user-defined return value semantics  
 PCLP-1800 Improved handling of enumeration constants of strongly typed `enums`  
 PCLP-1813 The `+efreeze` option now always prevents single-line suppressions  
 PCLP-1899 Don't issue 1564 when integer literal is converted/cast  
 PCLP-1911 Assume side-effect when passing pointer offset to non-const pointer parameter  
 PCLP-1932 Prototype information now included in message 1411  
 PCLP-1934 "Could be const" messages softened for typedefs from macros  
 PCLP-1951 Assume modification of initialized arguments when using `-fai`  
 PCLP-1956 Clarified message text for message 956  
 PCLP-1965 Add symbol information to message 9103  
 PCLP-1967 Issue message 564 for volatile reads  
 PCLP-1969 Improved handling of deallocation tracking in `if` statements  
 PCLP-1970 False positive 773 for named casts  
 PCLP-1971 Exempting logical not from message 1564  
 PCLP-1976 Improved performance for next-statement suppressions  
 PCLP-1980 1938 no long issued for `constexpr` variable  
 PCLP-1985 Improved handling of structures initialized by functions with using `fai`  
 PCLP-1987 Improved output format for `enum` essential types in essential type messages  
 PCLP-1992 False positive 522 in `__asm` statement  
 PCLP-2002 Message 785 softened for aggregate initialization with extra braces  
 PCLP-2006 Recognition of aggregate initialization evaluation order in C++11  
 PCLP-2024 Support flexible array members in base classes  
 PCLP-2039 Clarifications to text of message 9128  
 PCLP-2050 Improved `pclp_config` support for macro definition that contain spaces and quotes in Visual Studio configurations  
 PCLP-2074 Improved Value Tracking handling of `while` statement condition variable declarations  
 PCLP-2075 Improve application of dynamic allocation semantics  
 PCLP-2081 Message 1793 no longer issued for functions with an rvalue reference qualifier  
 PCLP-2088 Improved handling of premature termination due to stack overflow  
 PCLP-2098 Improved Value Tracking for unconditional assignment in both branches of an `if` statement  
 PCLP-2152 Improved handling of friends and templates for 9004  
 PCLP-2161 Improved handling of misspelled parameterized suppression options  
 PCLP-2164 Add symbol parameter to null pointer dereference messages when available  
 PCLP-2166 Message 904 issued multiple times for repeated declarations  
 PCLP-2178 Suppress message 1788 for variables marked as unused  
 PCLP-2185 Improved error handling for invalid size and alignment options  
 PCLP-2195 Improved handling of possibly null information in user-defined return semantics  
 PCLP-2200 Improved handling of buffers used with placement `new`  
 PCLP-2205 Improved support for C++17 `constexpr if`  
 PCLP-2214 Added `env-html.lnt`, `env-html.js`, and `env-xml.lnt` files

### 23.9.1.6 New Features

PCLP-1531 Support for IAR compilers and IAR Workbench  
 PCLP-1889 New message 3450 - subtracting member from 'this' pointer  
 PCLP-2051 New `fbe` flag option and backslash escapes  
 PCLP-2054 Added support for MISRA C 2012 AMD-1 Rule 12.5  
 PCLP-2055 Added support for MISRA C 2012 AMD-1 Rule 21.13  
 PCLP-2057 Added support MISRA C 2012 AMD-1 Rule 21.15  
 PCLP-2058 Added new message (9098) to support MISRA C 2012 AMD-1 Rule 21.16  
 PCLP-2059 Add partial support for MISRA C 2012 AMD-1 Rule 21.17

|           |                                                                        |
|-----------|------------------------------------------------------------------------|
| PCLP-2083 | Allow the C++17 fallthrough attribute to suppress messages 616 and 825 |
| PCLP-2132 | The <code>-help</code> option now responds to message numbers          |
| PCLP-2201 | New exit command added to Value Tracking debugger                      |

### 23.9.1.7 Documentation Enhancements

|           |                                                                                           |
|-----------|-------------------------------------------------------------------------------------------|
| PCLP-1926 | Add "Flow of Execution" section (1.2) to Reference Manual                                 |
| PCLP-1949 | Fix runaway text in Reference Manual                                                      |
| PCLP-1966 | Update description of message 9049                                                        |
| PCLP-2008 | Reference the <code>flf</code> flag in the Value Tracking section of the Reference Manual |
| PCLP-2012 | Improvements to descriptions of select error messages                                     |
| PCLP-2016 | <code>fwc</code> flag removed from manual                                                 |
| PCLP-2049 | Miscellaneous documentation corrections                                                   |
| PCLP-2172 | Clarify that <code>-egrep</code> doesn't match text injected via <code>+typename</code>   |
| PCLP-2210 | Update description of message 916 to remove MISRA C++ support statement                   |

## 23.10 Differences from PC-lint® 9.0

Note: This section describes differences between PC-lint 9.0 and the initial release of PC-lint Plus 1.0.

- PC-lint Plus contains many more diagnostics than PC-lint. As a result, the range of message numbers has increased. In particular, C language diagnostics extend into the 2000-2999 range, C++ diagnostics extend into the 3000-3999 range, the range 4000-5999 is used for new general error messages, and the range 8000-8999 are reserved for user-defined messages.. See [22 Messages](#) for complete details.  
Only message suppression options that appear before the location specified in the message are considered, even if the actual message is not issued until later (such as at global wrapup).
- K&R (traditional, pre-ANSI) C is no longer supported. In particular, options and flags which were specific to K&R C will not be supported. We will continue to support C89/C90, C99, and C11.
- The default diagnostic format has changed. By default, we now display the message first, followed by the corresponding source line. We also use a caret (^) to indicate source positions by default instead of an underscore. Finally, tildes (~) are used to underscore relevant parts of source code. The default PC-lint 9.0 format can be obtained using the option `-ha_3`.
- Macro display has changed. When a diagnostic is issued from a location which is the result of a macro expansion, a separate [893](#) note message will be attached to the original message for each expansion associated with the message.
- The message category emitted with each message (error, warning, info, note and supplemental) is now displayed in all lowercase letters by default which is a departure from PC-lint which capitalized the first letter. The new `+fcc` option can be used to revert to the previous behavior.
- The `-c` option is no longer the preferred way to configure PC-lint Plus for a particular compiler. The problem is that it is not always clear exactly what the `-c` option does for a particular compiler and compiler versions cannot be specified with the `-c` option. Compiler-specific configuration is supported through the use of compiler `.lnt` files (which has been the case for most compilers for a while anyway).
- The non-standard use of `sizeof` within a preprocessor statement is no longer supported. This has never been allowed by ANSI/ISO C or C++ although it is an extension in some older compilers. A work-around is provided with the `-pp_sizeof` option.
- Constant variables and enumeration constants are not supported inside of user-defined function semantics. Macros are still expanded and environment variables can now be used in function semantic specifications as well.
- The suppression context of the location cited in a message is now taken into consideration when determining whether to suppress the message, despite when the message is issued. For example, given:

```
int x;
//lint -e714
int y;
```

PC-lint would not issue message 714 for either `x` or `y` because the message is not issued until wrap up time at which point message 714 is suppressed. In PC-lint Plus, when we consider issuing the message for `x`, we will remember the message suppression state at the point in which `x` was declared (the location provided in the message) and give the message since at that location the message is not suppressed. The message will not be issued for `y` because by this point the context includes the suppression of message 714. In most cases this results in considerably more intuitive behavior. For example, it is now possible to suppress wrap-up messages using a single-line suppression at the location in which the message is given which was not possible before:

```
int x; //lint !e714
```

- Options appearing within source code will no longer have any effect outside of the containing translation unit. In other words, at the end of each module, the option state reverts back to what it was before the module was processed. This is a change from PC-lint where the effect of an option provided in one source file would "leak" into following source files. This was rarely the intended behavior and resulted in situations where analysis would be dependent upon the order in which modules were processed. Note in particular that global options (such as `-u` [unit checkout]) no longer have an effect when appearing inside of source modules.
- The behavior of the precompiled header feature has changed. In particular, multiple PCH files are supported in one project (but only one per module) and the way that PCH files are processed diverges from PC-lint 9. See [6.1 Precompiled Headers](#) for more information.
- PC-lint Plus returns zero in the absence of fatal or internal errors, i.e. the default return value no longer reflects the number of messages emitted. The `-frz` option will restore the previous behavior. See [3.2 Exit Code](#) for details.

### 23.10.1 Major New Features

- Full support for recent versions of C and C++ including C99, C11, C++11 and C++14.
- We now support Visual Studio 2013, Visual Studio 2015 and Visual Studio 2017.
- Improved support for gcc compiler extensions such as case ranges, locally declared labels, and labels as values.
- Value Tracking contains a number of improvements including structure member and pointer tracking, new and improved diagnostics, and a new architecture that allows for deeper analysis.
- Significantly improved location information provided with diagnostics.
- Improvements to the Semantics feature including user-defined semantics that can be applied to individual function overloads, identification of invalid semantics, and user-defined return semantic validation.
- Strong Type checking and Dimensional analysis provide more detailed information and can suggest corrections.
- Format string checking supports positional arguments and implements several new checks.
- A number of new diagnostics have been added while outdated diagnostics have been removed.

**23.10.2 General Diagnostic Changes**

- PC-lint 9.00k introduced the 9xxx message range for additional Elective Notes, currently used mainly for MISRA C 2012 and (since 9.00L) MISRA C++ messages. PC-lint Plus adds the new message ranges 2000-2999 for C diagnostics, 3000-3999 for C++ diagnostics. The complete set of ranges is provided in the table below.

| Range     | Description         | Warning Level |
|-----------|---------------------|---------------|
| 1-199     | C Syntax Errors     | 1             |
| 200-299   | Internal Errors     | 1             |
| 300-399   | Fatal Errors        | 1             |
| 400-699   | C Warnings          | 2             |
| 700-899   | C Informational     | 3             |
| 900-999   | C Elective Notes    | 4             |
| 1000-1199 | C++ Syntax Errors   | 1             |
| 1200-1299 | Internal Errors     | 1             |
| 1300-1399 | C++ Fatal Errors    | 1             |
| 1400-1699 | C++ Warnings        | 2             |
| 1700-1899 | C++ Informational   | 3             |
| 1900-1999 | C++ Elective Notes  | 4             |
| 2000-2199 | C Syntax Errors     | 1             |
| 2400-2699 | C Warnings          | 2             |
| 2700-2899 | C Informational     | 3             |
| 2900-2999 | C Elective Notes    | 4             |
| 3000-3199 | C++ Syntax Errors   | 1             |
| 3400-3699 | C++ Warnings        | 2             |
| 3700-3899 | C++ Informational   | 3             |
| 3900-3999 | C++ Elective Notes  | 4             |
| 4000-5999 | C and C++ Errors    | 1             |
| 8000-8999 | User Defined        | 3             |
| 9000-9999 | Misc Elective Notes | 4             |

*Note: Messages related to C may also appear while processing C++ source but C++ messages should not appear while processing C source code.*

- The precision of the location (line and column) associated with most messages has been significantly improved. For example, given the C source:

```
void f() {
    int i;
}
```

PC-lint 9.00L generates the diagnostic:

```

}
Warning 529: Symbol 'i' (line 2) not subsequently referenced
```

while PC-lint Plus generates:

```

warning 529: local variable 'i' declared in 'f' not subsequently referenced
int i;
^
```

- The verbiage of many existing messages has been clarified and/or elaborated so as to more quickly understand the point of the diagnostic.
- All diagnostics now start with a lowercase letter. Previously, most messages began with an uppercase letter but this was not consistent.
- The message category (error, warning, info, note) is now emitted in all lowercase letters by default. Previously, the first letter was capitalized in messages (e.g. Error vs error). The previous behavior can be obtained using the **+fcc** flag.
- When the location of a message is the result of a macro expansion, this fact is relayed with the new message **893** ("expanded from macro '*string*'"). In PC-lint, this information was relayed using a macro display line that was emitted above the source context line.

### 23.10.3 Value Tracking

Major improvements to value tracking include:

- A new value tracking model, which keeps track of more value information and without utilizing multiple passes. The result is often a deeper and more accurate analysis with diagnostics presented in a more lucid fashion.
- Tracking of structure members.

In the following example, PC-lint previously did not detect a division by 0 due to the general absence of structure member tracking (outside of the **this** object for member functions):

```
struct S { int x; };
int f(int i) {
    S s;
    s.x = 0;
    return i / s.x;
}
```

While PC-lint Plus issues the following:

```
test.cpp 6 warning 414: division by zero
    return i / s.x;
           ^ ~~~

test.cpp 6 supplemental 831: expression evaluates to 0
    return i / s.x;
           ^~~
```

- Tracking of pointers.

PC-lint now supports the tracking of pointer values where feasible. In the following example, PC-lint previously could not report a division by 0 message as the indirect modification through `pi` was not recognized as modifying `i`:

```
int f(int x) {
    int i = 1;
    int *pi = &i;
    *pi = 0;
    return x / i;
}
```

PC-lint Plus reports the issue:

```
warning 414: division by zero
    return x / i;
    ~
supplemental 831: expression evaluates to 0
    return x / i;
    ^
```

As another example, PC-lint Plus now recognizes the division by 0 in the following example by realizing the call to `x` is really a call to `f` that sets the static variable `i` to 0 by tracking the values of function pointers:

```
static int i = 1;
void f() { i = 0; }

int g() {
    auto x = f;
    x();
    return 1 / i;
}
```

- As the new value tracking system can perform deeper intramodule analysis of specific calls in a single pass, some use cases for the `-passes` option have been supplanted by the new `-vt_depth` option, which is used to specify the maximum call stack depth PC-lint Plus should recurse during specific walks within a module. The default value is 2. A higher value will result in deeper walking during specific walks without resulting in excess memory usage or additional time spent when there are no functions at a higher level (unlike the `-passes` option in previous versions, which always results in memory and CPU increase). The `-passes` option (an alias of the new name `-vt_passes` for consistency) still controls the number of passes in which all modules are re-parsed and intermodule calls are performed.
- Tracking of floating point values.  
PC-lint Plus now tracks floating point values, which extends the previous behavior that tracked only integral values.

#### 23.10.4 Semantics

The following enhancements have been added to the Semantics feature:

- User-defined return semantics are now validated for specific calls when the body of the function is available. Violations of the return semantic are reported via the new **2426** message. In the following example, the semantic `@p > 0` specifies that the pointer return value is never null. The implementation of this function contains a path that violates this semantic. PC-lint Plus will now report when such a path is taken causing the return value semantic to be violated:

```
//lint -sem(f, @p > 0) return value should never be null
void *f(int a, void *p) {
    if (a < 0)
        return 0;
    return p;
}

void g(void *p) {
    void *ptr = f(-1, p);
}
```

PC-lint Plus produces:

```
warning 2426: return value (NULL) of call to function 'f(int, void *)'
             conflicts with return semantic '(@p>0)'
             void *ptr = f(-1, p);
                     ^
```

This checking extends user-defined semantics from a set of assumptions about a function where no body exists to a contract validation mechanism when the body is available.

- Separate user-defined semantics are now supported for overloaded functions.  
A user-defined semantic can be applied to a specific function overload by including the function's argument list in the semantic specification. A user-defined semantic that does not include an argument list will apply to all overloads that do not have a semantic associated with the specific overload.
- Certain GCC-style attributes are now (by default) automatically translated to the corresponding function semantics. See the description of the new **fca** flag option for details.
- PC-lint Plus now warns when an invalid user-defined semantic is being rejected for a specific call via the new **2425** message. An explanation of why the semantic is being discarded is provided in the message. For example, the semantic below specifies that the first numeric argument to function **f** should be larger than the second argument. In the actual call, there is no second argument so the semantic doesn't apply:

```
//lint -sem(f, 1n > 2n)
int f(int i);
void g() {
    f(1);
}
```

PC-lint Plus will now emit the diagnostic:

```
warning 2425: user-defined function semantic '(1n>2n)' was
             rejected during call to function 'f(int)' because '2n'
             references non-existent argument in call
             f(1);
             ^
```

The warning highlights a likely mistake in the specification of the semantic. If there are multiple overloads and the semantic should only apply to certain overloads, the overload-specific semantic specification described above can be employed.

- PC-lint Plus favors the information in a user-defined return semantic even when more precise information was known about this value. The default behavior is to retain the more specific information. The old behavior can be reinstated using the new **+fso** flag option. This only applies to functions with an implementation visible to PC-lint.
- New Elective Notes **9901** and **9902** specify debugging information related to the information inferred from a return semantic for a specific call.

### 23.10.5 Strong Types and Dimensional Analysis

The messages related to strong type violations are more descriptive and can provide suggestions for certain erroneous constructs. In the following example:

```
//lint -strong(JcAc, Mi, Km, MiPerKm = Mi / Km)
typedef double Mi, Km, MiPerKm;
MiPerKm mi_per_km = 0.62137;

Mi earth_radius = 7918;
void f() {
    Km earth_circumference = 2 * 3.14159 * earth_radius;
}
```

PC-lint would previously generate the following diagnostic:

```
-
    Km earth_circumference = 2 * 3.14159 * earth_radius;
Warning 632: Assignment to strong type 'Km' in context: initialization
```

PC-lint Plus now generates the following:

```
warning 632: strong type mismatch: assigning 'Mi' to 'Km' in
context 'initialization'
    Km earth_circumference = 2 * 3.14159 * earth_radius;
                        ^~~~~~
supplemental 892: did you mean to divide by a factor of type 'MiPerKm'?
```

### 23.10.6 Improved Format String Checking

The checking of `printf` and `scanf` format strings has been significantly improved. In particular, PC-lint Plus now recognizes POSIX positional arguments and diagnoses issues related to positional arguments, existing messages have been re-organized and clarified, and new diagnostics have been added.

### 23.10.7 Miscellaneous enhancements and Quiet Changes

- The new `-std` option (e.g. `-std=c++14`) is now the preferred way of specifying a standard language version. The supported values for this option are: `c89/c90`, `c99`, `c11`, `C++03`, `C++11/C++0x`, `C++14/C++1y`, and `C++17/1z`. Unlike the `-A` option, this option requires one of the above values, e.g. neither `C++2011` nor `13` is a valid way to specify `C++11` support. The `-A` option is still supported but deprecated, users are encouraged to switch to the new `-std` option.
- The default C language version is `C11` and the default C++ language version is `C++14`. In previous versions the defaults were `C99` and `C++03` respectively. To restore the previous behavior, the options `-std=c99` and `-std=C++03` may be used.
- The `--e{` option now suppresses messages for the entire enclosing braced region, as the manual has always indicated. Previously, it did not apply to messages issued earlier in the braced region.
- A space-separated list of messages may be specified in parenthesized suppression options, e.g. `-esym(123 456 117*, A, B, C)` will suppress messages `123`, `456`, and `1170-1179` when parameterized by the symbol `A`, `B`, or `C`. This was a long-standing undocumented feature until 9.00L when it was inadvertently removed. The feature has been re-instated with official status.
- Environment variables surrounded by `%` are now supported in all options and in `.lint` files. Previously environment variables were only expanded in a few places.



### 23.10.8 Error Inhibition

`-egrep(# [#]..., regex [,regex]...)` inhibits the message *#s* when the message text matches *regex*  
`+egrep(# [#]..., regex [,regex]...)` enables the message *#s* when the message text matches *regex*

`+group(name [,pattern]...)` adds messages to a group

`-group(name [,pattern]...)` remove messages from a group or delete a grouping

The `-efile/+efile` options now suppress messages *within* the specified files instead of messages *about* the specified files.

### 23.10.9 Verbosity

`-verbosity(string)` print *string* as a verbosity message

### 23.10.10 Message Presentation

`+message([#,] text)` emits a custom message with the specified message *#*

### 23.10.11 Miscellaneous Options

#### 23.10.11.1 Global

`-cond(conditional-expr, true-options [,false-options])` conditionally execute options

`-dump_message_list=filename` dumps PC-lint Plus message list to the provided file

`-help=option` display detailed help about a specific *option*

`-write_file(string, filename [,append=true|false] [,binary=true|false])` write a *string* to a *filename*

`+zero_err(# [#]...)` specify message numbers that shouldn't increase exit code

`-zero_err(# [#]...)` specify message numbers that should increment exit code

#### 23.10.11.2 Output

`-env_push` push the current option environment

`-env_pop` pop the current option environment

`-env_save(Name)` save the current option environment

`-env_restore(Name)` restore the option environment to a previously saved one

#### 23.10.11.3 New Flag Options

`fb1` dependent base class lookup in templates (default OFF).

`fcc` capitalize message categories (default OFF).

`fce` continue on `#error` (default OFF).

`fcs` continue on static assertion failure (default OFF).

`fdg` expansion of digraphs (default ON).

`fdm` comma from macro expansion does not delimit macro args (default OFF).

`fdt` delayed template parsing (default OFF).

`fee` expand environment variables (default ON).

`fei` underlying type for Enum is always Int (default OFF).

`fes` search enclosing scopes for friend tag `decls` (default OFF).

`ffv` implicit function to void pointer conversion (default OFF).

`ffw` allow friend `decl` to act as forward `decl` (default OFF).

`fgi` inline treated as GNU inline (default OFF).

`fho` header include guard optimization (default OFF).

**fla** locations for all diagnostics (default ON).  
**flp** lax null pointer constants (default OFF).  
**fma** microsoft inline asm blocks (default OFF).  
**fms** microsoft semantics (default OFF).  
**fon** support for C++ operator name keywords (default ON).  
**frc** remove commas before `__VA_ARGS__` (default OFF).  
**frd** redefine default params for class template function members (default OFF).  
**fse** use smallest underlying type for enums (default OFF).  
**fsi** search `#include` stack (default OFF).  
**fso** return semantics override deduced return values (default OFF).  
**fum** user declared move deletes only corresponding copy (default OFF).  
**fur** allow unions to contain reference members (default OFF).

## 24 Open Source Declarations

PC-lint Plus incorporates several pieces of Open Source Software. The following declarations are made in compliance with the respective licenses.

- [LLVM/clang](#)
  - Attribution: C and C++ front-end support is provided by the LLVM and clang projects.
  - License: [Apache License, Version 2.0](#) with [LLVM Exceptions](#)
- OpenBSD regex
  - Attribution: The LLVM software incorporated into PC-lint Plus may utilize code from the OpenBSD regex library.
  - License:

Copyright 1992, 1993, 1994 Henry Spencer. All rights reserved. This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

- [vswhere](#)

- License:

The MIT License (MIT)  
Copyright (C) Microsoft Corporation. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF

MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- **PCRE**

- Attribution: Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England.
- License: PCRE2 is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Release 10 of PCRE2 is distributed under the terms of the "BSD" license, as specified below. The documentation for PCRE2, supplied in the "doc" directory, is distributed under the same terms as the software itself. The data in the testdata directory is not copyrighted and is in the public domain.

The basic library functions are written in C and are freestanding. Also included in the distribution is a just-in-time compiler that can be used to optimize pattern matching. This is an optional feature that can be omitted when the library is built.

#### **THE BASIC LIBRARY FUNCTIONS**

Written by: Philip Hazel Email local part: ph10 Email domain: cam.ac.uk  
University of Cambridge Computing Service, Cambridge, England.  
Copyright (c) 1997-2016 University of Cambridge All rights reserved.

#### **PCRE2 JUST-IN-TIME COMPILE SUPPORT**

Written by: Zoltan Herczeg Email local part: hzmester Email domain: freemail.hu  
Copyright(c) 2010-2016 Zoltan Herczeg All rights reserved.

#### **STACK-LESS JUST-IN-TIME COMPILER**

Written by: Zoltan Herczeg Email local part: hzmester Email domain: freemail.hu  
Copyright(c) 2009-2016 Zoltan Herczeg All rights reserved.

#### **THE "BSD" LICENSE**

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University of Cambridge nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS

OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 25 Acknowledgements

### 25.1 Carnegie Mellon University

Chapters 14 (guideline titles in Description column) and 22 (guideline titles in CERT C support statements) and configuration files for SEI CERT C checking (guideline titles in `au-certc.lnt`) incorporate portions of the “SEI CERT C Coding Standard Wiki” (<https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>), Copyright © 1995–2016 Carnegie Mellon University, with special permission from its Software Engineering Institute.

ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Neither the portions of this distribution incorporating portions of the “SEI CERT C Coding Standard Wiki” nor any other portion of this distribution nor any other distribution by Vector Informatik GmbH has been reviewed nor endorsed by Carnegie Mellon University or its Software Engineering Institute.

CERT is a registered trademark of Carnegie Mellon University.

### 25.2 MITRE Corporation

Chapters 15 (weakness names in Name column) and 22 (weakness names in CWE support statements) and configuration files for CWE checking (weakness names in `au-cwe-full.lnt`) incorporate portions of the “CWE Common Weakness Enumeration” (<https://cwe.mitre.org>), Copyright © 2006–2023 The MITRE Corporation.

#### Terms of Use

CWE is free to use by any organization or individual for any research, development, and/or commercial purposes, per these CWE Terms of Use. The MITRE Corporation ("MITRE") has copyrighted the CWE List, Top 25, CWSS, and CWRAF for the benefit of the community in order to ensure each remains a free and open standard, as well as to legally protect the ongoing use of it and any resulting content by government, vendors, and/or users. CWE is a trademark of MITRE. Please contact [cwe@mitre.org](mailto:cwe@mitre.org) if you require further clarification on this issue.

#### LICENSE

CWE Usage: MITRE hereby grants you a non-exclusive, royalty-free license to use CWE for research, development, and commercial purposes. Any copy you make for such purposes is authorized on the condition that you reproduce MITRE’s copyright designation and this license in any such copy.

#### DISCLAIMERS

ALL DOCUMENTS AND THE INFORMATION CONTAINED IN THE CWE ARE PROVIDED ON AN "AS IS" BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE INFORMATION OR THE USE OR OTHER DEALINGS IN THE CWE.

## 26 Bibliography

- [1] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.
- [2] Dan Saks. *const T vs. T const*. [www.dansaks.com](http://www.dansaks.com), Published Articles, 1999.
- [3] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates – The Complete Guide*. Addison-Wesley.
- [4] ISO/IEC. *14882:2003 Programming Languages C++*. International Standard, 2003.
- [5] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003.
- [6] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.
- [7] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.
- [8] David A. Spuler. *C++ and C Debugging, Testing and Reliability*. Prentice Hall, 1994.
- [9] Scott Meyers. *Effective C++ Third Edition*. Addison-Wesley.
- [10] ISO/IEC. *14882 C++ Standard Core Language Defect Reports and Accepted Issues*.
- [11] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards (101 Rules, Guideline, and Best Practices)*. Addison-Wesley.
- [12] Allen I. Holub. *Enough Rope to Shoot Yourself in the Foot*. McGraw Hill, 1995.
- [13] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, First Printing 1990, Reprint w/corrections 1992.
- [14] ISO/IEC. *14882:1998 - Programming Languages - C++*. American National Standards Institute, 1998.
- [15] Tom Cargill. *C++ Gotchas*. Presented at C++ World, November 1992.
- [16] Andrew Koenig. *Check list for Class Authors*. 1992 Nov 1.
- [17] Tom Cargill. *C++ Programming Style*. Addison-Wesley, 1992.
- [18] The Motor Industry Research Association. *Guidelines of the Use of the C Language in Vehicle Based Software (MISRA)*. Warwickshire, 1998.
- [19] Dan Saks. *C++ Gotchas!* Saks and Associates.
- [20] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley.
- [21] Herb Sutter. *Exceptional C++*. Addison-Wesley.
- [22] Scott Meyers. *Effective Modern C++*. O'Reilly, 2015.
- [23] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 1978.
- [24] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Pearson, 5th edition, 2002.
- [25] ISO/IEC. *9899:1999 Programming languages - C*. American National Standards Institute, 1999.
- [26] INCITS/ISO/IEC. *14882-2011 Programming languages - C++*. American National Standards Institute, 2012.
- [27] Robert Ward. *Debugging C*. Que Corporation, 1986.
- [28] Rex Jaeschke. *Portability and the C Language*. Hayden Books, 1989.
- [29] Les Hatton. *Safer C*. McGraw-Hill, 1995.
- [30] Peter Van Der Linden. *Expert C Programming - Deep C Secrets*. Prentice Hall, 1994.



- [31] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.
- [32] Bruce Eckel. *C++ Inside and Out*. Osborne / McGraw-Hill, 1992.
- [33] S. Hekmatpour. *C++: A Guide for Programmers*. Prentice Hall, 1992.
- [34] T. Plum and D. Saks. *C++ Programming Guide*. Plum Hall, 1991.
- [35] Bjarne Stroustrup. *The C++ Programming Language., 2nd Ed.* Addison-Wesley, 1992.
- [36] Larry Reznick. *Tools for Code Management*. R&D Books, 1996.
- [37] Andrew Koenig and Barbara Moo. *Ruminations on C++*. Addison-Wesley.
- [38] Bil Lewis and Daniel J. Berg. *Multithread Programming with Pthreads*. Sun Microsystems Press.
- [39] The Motor Industry Research Association. *MISRA-C:2004 Guidelines for the use of the C Language in critical systems*. 2004.
- [40] The Motor Industry Software Reliability Association. *MISRA-C++:2008 Guideline for the use of the C++ Language in critical systems*. The Motor Industry Research Association, 2008.

