PC-lint/FlexeLint 9.0
Manual Excerpts

1. BASIC FACTS 1
2. INTRODUCTION 2
    2.1 An Example 2
    2.2 A Lint for C++ 3
    2.3 Language Definition 4
3. GETTING STARTED WITH PC-LINT 6
    3.1 Setup 6
    3.2 Configuration 7
    3.3 Running the Test Programs 8
    3.4 Linting your Programs 11
        3.4.1 Other File Extensions for C++ Modules 11
        3.4.2 Controlling the Messages 12
        3.4.3 Options 12
        3.4.4 Extending the Command Line 12
    3.5 Integrating With Your Environment 13
    3.6 Project Files 14
    3.7 DOS and OS/2 14
        3.7.1 Multiple Configurations 15
        3.7.2 DOS-ins Files 16
        3.7.3 DOS Extender Notes 18
4. THE COMMAND LINE 20
    4.1 Indirect (.lnt) Files 21
    4.2 Exit Code 21
5. OPTIONS 23
    5.1 Rules for Specifying Options 23
    5.2 Error Inhibition Options 26
        5.2.1 Meta Characters for -esym, -efile, -emacro, -efunc, -estring, -etype, -ecall 42
    5.3 Size and Alignment Options 46
    5.4 Verbosity Options 49
    5.5 Flag Options 51
    5.6 Message Presentation Options 80
        5.6.1 Message Height Option 80
        5.6.2 Message Width Option 83
        5.6.3 Message Format Options 83
        5.6.4 Appending Text to Messages 86
    5.7 Other Options 87
    5.8 Compiler Adaptation 121
        5.8.1 Microsoft Keywords 123
        5.8.2 Compiler Codes 124
        5.8.3 Customization Facilities 128

5.8.4 Identifier Characters 137
5.8.5 Preprocessor Statements 137
5.8.6 In-line assembly code 137
5.8.7 Pragmas 138
5.8.8 The General Solution 143
5.9 Self-Referencing Options Files 144
6. LIBRARIES 147
6.1 Library Header Files 147
6.2 Library Modules 151
6.3 Library Object Modules 152
6.4 Assembly Language Modules 153
7. FAST HEADER PROCESSING 155
7.1 Pre-compiled Headers 155
7.1.1 Introduction to pre-compiled headers 155
7.1.2 Designating the pre-compiled header 155
7.1.3 Monitoring pre-compiled headers 156
7.1.4 The use of make files 156
7.2 Bypass Headers 157
7.2.1 Constraints on Bypass Headers 160
8. LINT OBJECT MODULES 163
8.1 What is a LOB? 163
8.2 Why are LOB's used? 163
8.3 Producing a LOB 165
8.4 -lobbase to reduce lob sizes 165
8.5 Make Files 167
8.6 Library Modules 168
8.7 Options for LOB's 169
8.8 Limitations of LOB's 169
9. STRONG TYPES 170
9.1 Quick Start 170
9.2 What are Strong Types? 170
9.3 -strong 171
9.4 Multiplication and Division of Strong Types 175
9.4.1 Dimension (Jd) 176
9.4.2 Dimensionally Neutral (Jn) 176
9.4.3 Anti-Dimensional (Ja) 177
9.4.4 Dimensional Analysis 177
9.4.5 Conversions 179
9.4.6 Integers 180
9.4.7 Migrating to Dimensions 182
9.5 -index 183
9.6 Type Hierarchies 186
9.6.1 The Need for a Type Hierarchy 186
9.6.2 The Natural Type Hierarchy 186
9.6.3 Adding to the Natural Hierarchy 188

9.6.4 Restricting Down Assignments (-father) 189
9.6.5 Printing the Hierarchy Tree 190
9.7 Hints on Strong Typing 191
9.8 Reference Information 193
9.9 Strong Types and Prototypes 194
10. VALUE TRACKING 195
10.1 Initialization Tracking 195
10.2 Value Tracking 198
10.2.1 The assert remedy 201
10.2.2 Interfunction Value Tracking 203
10.2.3 Tracking Static Variables 213
11. SEMANTICS 217
11.1 Function Mimicry (**`-function`**) 217
11.1.1 Special Functions 217
11.1.2 Function listing 221
11.2 Semantic Specifications (**`-sem`**) 227
11.2.1 Possible Semantics 227
11.2.2 Semantic Expressions 236
11.2.3 Notes on Semantic Specifications 240
12. MULTI-THREAD SUPPORT 242
12.1 Overview 242
12.2 Identifying Threads 242
12.3 Mutual Exclusion 244
12.4 Thread-Protected (TP) Regions 246
12.5 Constructor-triggered mutex locking 247
12.6 Function Pointers 248
12.7 Thread Unfriendly Functions 249
12.7.1 Thread Unsafe Functions (Category 1) 249
12.7.2 Category 2 Functions 251
12.7.3 Category 3 Functions 252
12.7.4 Header Options 252
12.7.5 Directory Options 253
12.7.6 Thread Unsafe Classifications 253
12.7.7 Priorities in Thread Unsafety 254
12.7.8 Category 4 Functions 255
12.7.9 Category 5 Functions 256
12.8 Thread Local Storage 256
12.8.1 __thread 256
12.8.2 __declspec(thread) 256
12.9 Atomic Access 257
12.9.1 Atomic Operations 257
12.9.2 Atomic Types 258
12.10 Declarative Methods 260
13. OTHER FEATURES 263

13.1 Order of Evaluation ... 263
13.2 Format Checking ... 265
13.3 Indentation Checking ... 265
13.4 const Checking ... 268
13.5 volatile Checking ... 268
13.6 Prototype Generation ... 269
13.7 Exact Parameter Matching ... 271
13.8 Weak Definials ... 274
13.8.1 Unused Headers ... 274
13.8.2 Removable from Header ... 276
13.8.3 static-able ... 276
13.9 Unix Lint Options ... 277
13.10 Static Initialization ... 278
13.11 Size of Scalars ... 279
13.12 MISRA Standards Checking ... 280
13.13 Stack Usage Report ... 280
14. NON-STANDARD EXTENSIONS ... 285
14.1 Memory Models ... 285
14.2 Additional Reserved Words ... 286
15. PREPROCESSOR ... 287
15.1 Preprocessor Symbols ... 287
15.2 include Processing ... 288
15.2.1 INCLUDE Environment Variable ... 289
15.3 ANSI/ISO Preprocessor Facilities ... 289
15.3.1 #line and # ... 289
15.4 Non-Standard Preprocessing ... 290
15.4.1 #assert ... 290
15.4.2 #c_include ... 290
15.4.3 #asm ... 291
15.4.4 #dictionary ... 291
15.4.5 #endasm ... 291
15.4.6 #import ... 291
15.4.7 #unassert ... 292
15.4.8 #include_next ... 292
15.5 User-Defined Keywords ... 293
16. LIVING WITH LINT ... 294
16.1 An Example of a Policy ... 294
16.2 Recommended Setup ... 296
16.3 Using Lint Object Modules ... 297
16.4 Summarizing ... 298
17. PROGRAM INFORMATION ... 299
17.1 Record Fields ... 300
17.1.1 The **file** category (*Prefix***file.txt**) ... 300
17.1.2 The **type** category (*Prefix***type.txt**) ... 301
17.1.3 The **symbol** category (*Prefix***symbol.txt**) ... 301

17.1.4 The `macro` category (*`Prefix`*`macro.txt`) 302
17.2 Output Format Strings 302
17.3 Enabling and Suppressing Output 304
17.4 Flag fields 305
17.4.1 File Flags 305
17.4.2 Symbol Flags 306
17.4.3 Macro Flags 306
17.4.4 Output All Flags 307
17.5 Output Filtering 307
18. COMMON PROBLEMS 309
18.1 Option has no effect 309
18.2 Order of option processing 309
18.3 Too many messages 309
18.4 What is the preprocessor doing? 310
18.5 NULL not defined 310
18.6 Error 123 using min or max 310
18.7 LONG_MIN macro 311
18.8 Plain Vanilla Functions 311
18.9 Avoiding Lint Comments in Your Code 312
18.10 Strange Compilers 313
18.11 !0 313
18.12 What Options am I using? 313
18.13 How do I deal with SQL? 314
18.14 Torture Testing Your Code 314
19. MESSAGES 316
19.1 C Syntax Errors 320
19.2 Internal Errors 335
19.3 Fatal Errors 335
19.4 C Warning Messages 339
19.5 C Informational Messages 388
19.6 C Elective Notes 416
19.7 C++ Syntax Errors 431
19.8 Additional Internal Errors 444
19.9 C++ Warning Messages 444
19.10 C++ Informational Messages 462
19.11 C++ Elective Notes 482
20. WHAT'S NEW 493
20.1 Major New Features 493
20.2 New Error Inhibition Options 497
20.3 New Verbosity Options 498
20.4 New Flag Options 498
20.5 New Message Presentation Options 499
20.6 Additional Other Options 500
20.7 Compiler Adaptation 501
20.8 New Messages 502
21. BIBLIOGRAPHY 508

5.8.3 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/FlexeLint to ignore some nonstandard constructs accepted by some compilers.

`@` Compilers for embedded systems frequently use the `@` notation to specify the location of a variable. A technique to handle this is given in our manual in Section 5.8.2 Compiler Codes in the description of **`-cwh`**. Users have encountered some difficulty with this method when the location is given as a complex expression. We have for this reason added direct support for the `@` feature, which consists of ignoring expressions to its right. To enable this, just make sure you are NOT using **`-cwh`**. 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`**.

`_bit` is a type that is one bit wide. This needs to be activated with the **`+rw(_bit)`** option. It was introduced to support some microcontroller cross-compilers that have a one-bit type.

`_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/FlexeLint 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/FlexeLint 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/FlexeLint 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/FlexeLint 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, which you want PC-lint/FlexeLint 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.

-a#*predicate*(*token-sequence*)

asserts the truth of ***#predicate*** for the given ***token-sequence***. This is to support the Unix System V Release 4 **#assert** facility. For example:

```
          -a#machine( pdp11 )
```

makes the predicate **#machine(pdp11)** true. See also Section 15.4 Non-Standard Preprocessing.

+/-compiler(*flag1* [,*flag2* ...]) This option allows the programmer to specify flags that describe compiler-specific behavior. As of this writing this option takes the following flags:

base_op (OFF by default) -- This flag changes the meaning of the digraph token "**:>**". This flag causes "**:>**" to be interpreted as an operator whose LHS represents a segment and whose RHS represents an offset within that segment. This usage was introduced by an earlier version of the Microsoft C compiler and was useful to support segmented architectures then in popular use. Since the introduction of 32 bit compilers this has become less frequently used. For example,

```
        +compiler( base_op )
```

enables this meaning of "**:>**".

Note: According to the C99 and C++ Standards, this token is a synonym for "**]**" making this older interpretation really obsolete.

std_alt_keywords (OFF by default) -- Enables C++ standard alternative keywords (e.g., "**and**" is a synonym for "**&&**"). The standard keys include: **and**, **bitor**, **or**, **xor**, **compl**, **bitand**, **and_eq**, **or_eq**, **xor_eq**, **not**, **not_eq**. For example,

```
        +compiler( std_alt_keywords )
```

enables standard alternative keywords.

std_digraphs (OFF by default) -- Enables the interpretation of the C99/C++ digraph tokens "**<:**" and "**:>**". For example,

```
        +compiler( std_digraphs )
```

enables the standard meaning of these two digraphs.

This is off by default because the following code, although technically ill-formed, is often permitted by default by most compilers:

```
    struct A{};
    template< class T > struct B{};
```

```
::B<::A> z; // syntax error: equivalent to " :: B [ : A > z ;"
```

(Note: since this kind of thing can't happen with the alternative digraphs "**<%**" and "**%>**", they are always enabled.)

-d*name*{*definition*} is an alternative to **-d*name*=*definition***.

-d*name*{*definition*} has the advantage that blanks may be embedded in the definition. Now its 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.

One such condition is output produced by the scavenger whose purpose is to extract pre-defined macro definitions from an unwitting compiler. See **-scavenge** for details.

-d*name*()=*Replacement*

-d*name*(*identifier-list*)=*Replacement*

To induce PC-lint/FlexeLint 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_varying(n)
```

as a type and you want to get PC-lint/FlexeLint 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.

`-#d`*`name`*`=`*`Replacement`*
This is yet another variation on the global define facility. It affects only **`#include`** lines and is intended to support the VAX-11 C includes. For example:

```
            #include   time
```

is supported by a **`-#dtime=`*`Filename`*** option and does not affect any other uses of the **`time`** identifier.

`-overload(`*`X`*`)` will set flags, which can affect function overload resolution. This option is highly technical but may be required to resolve some very subtle overload resolution incompatibilities among different compilers. ***`X`*** is a hexadecimal number (without the leading '**`0x`**'). For example, **`-overload(5)`** sets bits 1 and 4. The bits have the following meaning.

1 Memory model counts more than ANSI/ISO qualification. For example, if this flag is set **`int n; f(&n);`** chooses **`void f(int const *)`** over **`void f(int far *)`** because **`far`** 'outweighs' the **`const`**.

2 Memory model plus ANSI/ISO qualification exceeds either alone. For example, if this flag is set **`int n; f(&n);`** chooses **`void f(int const *)`** over **`void f(int const far *)`** rather than regard the call as ambiguous.

4 Memory model has significance for references. For example, if this flag is set, **`int n; f(n);`** chooses **`void f(int &)`** over **`f(int far &)`** because the **`far`** has significance with references.

8 Memory model ignored when declaring **`operator delete`**. With this flag off (the default) it is possible to distinguish, for example, between the following two declarations:

```
        void operator delete( void * );
        void operator delete( void far * );
```

The default is **`-overload(7)`**.

The compiler selection flags for Microsoft (**`-cmsc`**) and Borland (**`-ctc`**) automatically adjust this set of flags (to 7 and 2 respectively).

Memory model differences are actual not nominal. For example, **`char far *`** is not considered different from **`char *`** in the large memory model (**`-mL`**). Memory model differences only relate to pointer sizes or the implied pointer of a reference. For example, passing a **`far int`** to an **`int`** requires no conversion.

-plus(***char*** **)** identifies ***char*** as an alternate '**+**' character used for options. If it is difficult to use the '**+**' character on the command line you may use an alternate character specified by this option. E.g., **-plus(&)**.

-scavenge(***filename-pattern*** [, ...] **)**
-scavenge(clean, ***filename*** **)**

The purpose of this option is to automatically find a compilers' built-in macros. It completely changes the character of Lint from static analyzer to a scavenger of macros (or cleanup facility depending on the sub-option).

Users of retargetable and embedded compilers may find difficulty in configuring Lint to parse compiler-provided and 3rd-party headers correctly. The main reason is that such compilers tend to make extensive use of pre-defined macros (that is, macros for which no definition exists in any header file). To make matters worse, these compilers do not always provide a way to dump a list of macro definitions. For example, the GCC compiler provides the pair of options "**-E -dM**" which will dump the macros that are pre-defined into a file, in a form that can be used directly by Lint. This file can be cited in a **-header** option.

For any given compiler, you can get a list of macro definitions by using the four-step process below:

The option has two modes. The first is of the form:

-scavenge(***filename-pattern*** [, ...] **)**

and the second is of the form:

-scavenge(clean, ***filename*** **)**

The first mode is for generating and the second mode is for cleaning up. Here are the steps.

1) Tell Lint where to look for headers by providing a set of **-i** options (or set the INCLUDE environment variable). The directories named by these options should be those that the compiler searches by default (e.g. Standard library headers). For this example, assume we have placed such a set of **-i** options in the file "**include.lnt**"

2) Run:

```
lint include.lnt -scavenge(*.h) >mac.c
```

The actual command name varies according to the system. C++ users will probably want to use a ***filename-pattern*** of "*****" rather than "***.h**" and an extension of "**.cpp**" rather than "**.c**"

For each unique identifier found in any of the files matching the argument(s) of the options **`-scavenge`** you will obtain in the output a 3-line sequence of the form:

```
#ifdef name
-dname{name}
#endif
```

where ***`name`*** is the name of the identifier. Thus, for example, you will probably see (among thousands of other 3-line sequences) the following:

```
#ifdef __cplusplus
-d__cplusplus{cplusplus}
#endif
```

This is not valid C or C++ but it can be passed through your compiler's preprocessor. If you were to use **`mac.c`** as shown above then the compiler will ignore this sequence since the symbol **`__cplusplus`** is not defined for C. However if you were to have used **`mac.cpp`** then the compiler's preprocessor will probably produce as output:

```
-d__cplusplus {1}
```

because for most C++ compilers **`__cplusplus`** is defined to be **`1`** and **`d__cplusplus`** is simply not defined.

3) Run "**`mac.c`**" through your compiler's preprocessor and capture the output in a file which you should designate as a "**`.lnt`**" file. We arbitrarily pick "**`mac.lnt`**" here. For example, with MSVC7, this means running:

```
cl [normal compiler options] /EP mac.c >mac.lnt
```

Now **`mac.lnt`** contains a complete set of all pre-defined macros used in the compiler's headers but will, in all likelihood, contain numerous blank lines. These can be cleaned up by using the scavenge in '**`clean`**' mode.

4) run:

```
lint -scavenge(clean,mac.lnt)
```

In **`clean`** mode, the scavenge option will open the file for reading, save the non-vacuous lines, close the file, open it up for writing and dump out the new contents.

Note: we are using Microsoft here as an example of how you might retrieve pre-processed output. It is not normally necessary to employ the scavenge option when using the Microsoft compiler as the necessary macros have already been extracted and placed in various **`co-msc*.lnt`** files.

Note: Whenever your build options change you may want to repeat steps 3 and 4.

`-template( X )` set the template flags to ***`X`***
`++template( X )` OR ***`X`*** into the template flags
`--template( X )` AND ~***`X`*** into the template flags where ***`X`*** is a hexadecimal constant specifying flags. This allows for fine-tuning of the template processing mechanism. Current flags are as follows:

1 aggressively process template base classes. Normally, base classes of class templates need not be processed until instantiation time. For some libraries, notably STL, base classes need to be aggressively processed because they supply names needed during the processing of the template itself.

2 When a template refers to itself recursively we normally presume this to be a self-reference or a mistake and in order to prevent run-aways, recursion is normally prohibited. The option **`-template(2)`** can be used to activate recursive template processing. The Rogue Wave library, for example, employs recursive template evaluation to implement two to the power of ***`N`***. Therefore, this option has been placed into the file: **`lib-rw.lnt`**

80 normally, template class member functions defined within the class are not instantiated unless referenced. This flag will force the instantiation of these in-line function.

100 Refrain from instantiating non-dependent template-id's. In the past we did not always instantiate these template-id's during a template definition. Currently we are more aggressive in doing this instantiation (up to the limits required by the language). Not all compilers (or even configurations of a particular compiler) instantiate identically. Activating this flag provides less aggressive behavior.

200 Examine dependent and formerly dependent base classes during unqualified name lookup. Section 14.6.2, paragraph 3 of the 2003 version of the ISO C++ Standard states

"In the definition of a class template or a member of a class template, if a base class of the class template depends on a template-parameter, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member."

However, during unqualified name lookup, some popular compilers do search in dependent (and formerly dependent) base classes both at template definition and at template instantiation time. To enable such name lookup behavior in Lint, use **`++template(200)`**. This flag will be ON by default when **`-cmsc`** or **`-cbc`** are given.

Note that MSVC7.1 does adhere to the Standard in this regard when given the **`/Za`** flag. Therefore, if you compile with **`/Za`**, you should probably also add **`--template(200)`** after **`-cmsc`** in your Lint configuration. Users of more recent versions of the Borland compiler may also choose to disable this **`-template`** bit.

400 Extend scopes of primary template parameters to specializations. Consider the following case:

```
template<class T> struct A;
template<> struct A<int> { T n; };
```

According to section 14.6.1, paragraph 3 of the 2003 ISO C++ Standard, "The scope of a template-parameter extends from its point of declaration until the end of its template." So in this example, "**`T`**" is not in scope after the semicolon that terminates the definition of the primary template of **`A`**. Furthermore, there is nothing to indicate that it is introduced in the scope of **`A<int>`**. However, some compilers (for example, versions 6 and 7 of the Microsoft compiler, as well as some other compilers in a backwards-compatibility mode) behave as if the template parameter **`T`** had been re-declared at the onset of **`A<int>`**. To enable similar behavior in Lint, use **`++template(400)`**. This bit is set automatically when **`-cmsc`** is used.

7. FAST HEADER PROCESSING

7.1 Pre-compiled Headers

7.1.1 Introduction to pre-compiled headers

Most readers of this information will already be familiar with the notion of a pre-compiled header. A single header is designated as one to be pre-compiled. When an include for this header is encountered, a special lookup is done to see if the header had been seen before. If it had not, then the header and all of its inclusions to whatever depth, are processed normally and the resulting digested form is deposited into a file using a name derived in some fashion from the original.

If the name of the header is **`x.h`**, the binary information will be dumped into **`x.lph`**. The 3-letter extension stands for Lint Precompiled Header.

If the header had been seen before, as evidenced by the existence of the file with the derived name, then in lieu of scanning the file, the deposited information is read in to create a state identical to the normal scanning process.

This scenario is pretty much replicated in this tool, at least for the first module. For subsequent modules that reference the original header, even this binary information is not read in since it may have been made obsolete while scanning the second module. Rather, most of the symbols that would have been generated by the header need simply to be made active rather than inactive. This is called bypassing and is actually a much faster process than reading in the pre-compilation.

The actual mechanics of bypassing will be discussed later in this chapter. It will be sufficient here to note that a header that is designated as a **`pch`** header will automatically be designated as a bypass header and this will be evident in the file verbosity (**`-vf`**).

7.1.2 Designating the pre-compiled header

To designate that a header is to be pre-compiled 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 pre-compiled 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 pre-compiled header. So earlier headers are permitted. But if a pre-compiled 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.

9. STRONG TYPES

9.2 What are Strong Types?

Have you ever gone through the trouble of making sure that your types are given appropriate typedef names and then wondered whether it was worth the trouble? It didn't seem like the compiler was checking these types for strict compliance.

Consider the following typical example:

```
typedef int Count;
typedef int Bool;
Count n;
Bool stop;
...
n = stop ; // mistake but no warning
```

This programmer botch goes undetected by the compiler because the compiler is empowered by the ANSI/ISO standards to check only underlying types, which, in this case, are both the same (**`int`**).

The**-strong** option and its supplementary option **-index** exist to support full or partial **typedef-**based type-checking. We refer to this as ***strong*** type-checking. In addition to checking, these options have an effect on generated prototypes. See Section 9.9 Strong Types and Prototypes

9.4 Multiplication and Division of Strong Types

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.

9.4.1 Dimension (Jd)

A ***dimension*** is a strong type such that when two expressions are multiplied or divided (including the modulus operator **%**) 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).

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 and so it is probably wise to place the **'d'** in explicitly.

Dimensional types are treated in greater detail later.

10. VALUE TRACKING

10.2 Value Tracking

Value Tracking was introduced with Version 7.0 of PC-lint/FlexeLint. By value tracking we mean that some information is retained about automatic variables (and about data members of the **this** class for a member function and, by Version 9.0, static variables) across statements in a fashion similar to what is retained about the state of initialization. (See Section 10.1 Initialization Tracking) Consider a simple example:

```
int a[10];

int f()
    {
    int k;

    k = 10;
    return a[k];    // Warning 415
    }
```

This will result in the diagnostic message Warning **415** (access of out-of-bounds pointer by operator '`[`') because the value assigned to `k` is retained by PC-lint/FlexeLint and used to decide on the worthiness of the subscript.

If we were to change things slightly to:

```
int a[10];

int f( int n )
    {
    int k;

    if ( n ) k = 10;
    else k = 0;
    return a[k];        // Warning 661
    }
```

we would obtain Warning **661** (Possible access of out-of-bounds pointer). The word 'possible' is used because not all of the paths leading to the reference `a[k]` would have assigned 10 to `k`.

Information is gleaned not only from assignment statements and initializations but also from conditionals. For example:

```
int a[10];

int f( int k, int n )
    {
    if ( k >= 10 ) a[0] = n;
    return a[k];            // Warning 661 -- k could be 10
    }
```

also produces Warning **`661`** based on the fact that **`k`** was ***`tested`*** for being greater than or equal to 10 before its use as a subscript. Otherwise, the presumption is that **`k`** is OK, i.e. the programmer knew what he or she was doing. Thus the following:

```
int a[10];

int f( int k, int n )
    { return a[k+n]; }      // no warning
```

produces no diagnostic.

Just as it is possible for a variable to be conceivably uninitialized (See Section 10.1 Initialization Tracking) it is possible for a variable to conceivably have a bad value. For example, if the loop in the example below is taken 0 times, **`k`** could conceivably be out-of-range. The message given is Informational **`796`** (Conceivable access of out-of-bounds pointer).

```
int a[10];

int f(int n, int k)
    {
    int m = 2;

    if( k >= 10 ) m++;     // Hmm -- So k could be 10, eh?
    while( n-- )
        { m++; k = 0; }
    return a[k];          // Info 796 - - k could still be 10
    }
```

In addition to reporting on the access of out-of-bounds subscripts (messages **`415`**, **`661`**, **`796`**), value tracking allows us to give similar messages for the division by 0 (**`54`**, **`414`**, **`795`**), inappropriate uses of the NULL pointer (**`413`**, **`613`**, **`794`**), the creation of illegal pointers (**`416`**, **`662`**, **`797`**) and the detection of redundant Boolean tests (**`774`**), as the following examples show.

Value tracking allows us to diagnose the possible use of the NULL pointer. For example:

```
int *f( int *p )
    {
    if ( p ) printf( "\n" );    // So -- p could be NULL
    printf( "%d", *p );         // Warning
    return p + 2;               // Warning
    }
```

will receive a diagnostic for the possible use of a NULL pointer in both the indirect reference (**`*p`**) and in the addition of 2 (**`p+2`**). Clearly both of these statements should have been within the scope of the **`if`**.

To create truly bullet-proof software you may turn on the Pointer-parameter-may-be-NULL flag (`+fpn`). This will assume the possibility that all pointer parameters to any function may be NULL.

Bounds checking is done only on the high side. That is:

```
  int a[10]; ...  a[10] = 0;
```

is diagnosed but `a[-1]` is not.

There are two sets of messages associated with out-of-bounds checking. The first is the creation of an out-of-bounds pointer and the second is the access of an out-of-bounds pointer. By "access" we mean retrieving a value through the pointer. In ANSI/ISO C **[1] 3.3.6**) you are allowed to create a pointer that points to one beyond the end of an array. For example:

```
  int a[10];
  f( a + 10 );        // OK
  f( a + 11 );        // error
```

But in neither case can you access such a pointer. For example:

```
  int a[10], *p, *q;
  p = a + 10;         // OK
  *p = 0;             // Warning (access error)
  p[-1] = 0;          // No Warning
  q = p + 1;          // Warning (creation error)
  q[0] = 0;           // Warning (access error)
```

As indicated earlier, we do not check on subscripts being effectively negative. We check only on the high side of an array.

Though not as critical as pointer checking, tracking values allows us to report that a Boolean condition will always be true (or false). Thus

```
     if ( n > 0 ) n = 0;
     else if ( n <= 0 ) n = -1;      // Info 774
```

results in the Informational message **`774`**) that the second test can be ignored. Such redundant tests are usually benign but they can be a symptom of faulty logic and deserve careful scrutiny.

10.2.2 Interfunction Value Tracking

With interfunction value tracking PC-lint/FlexeLint will keep track of values that are passed to functions. When the definition of a called function is encountered, such values are then used to initialize the values of parameters. This can be used to determine return values, to record additional function calls and, of course, to detect errors. To take a very blatant example consider the following module:

```
    t1.cpp:

  1     int f(int);
  2     int g()
  3         { return f(0); }
  4     int f( int n )
  5         { return  10 / n; }
```

In this example, **`f()`** is called with an argument of 0. This turns the innocent looking **`10/n`** into a lethal divide by 0.

With the command **`lin -u t1.cpp`** we get the following output:

```
  --- Module:   t1.cpp

  During Specific Walk:
    File t1.cpp line 3: f(0)
  t1.cpp  5  Warning 414: Possible division by 0 [Reference:
      File t1.cpp: line 3]
```

The first thing you notice is the phrase "During Specific Walk". (The notion of Specific Walk is defined more formally in the next section). This is then followed by the location of a function call, the name of the function ('**`f`**' in this case) and a description of the arguments ('**`0`**' in this case). This is then followed by a conventional error message. Missing from the message is a reproduction of the line in error and a handy cursor to identify where on the line the error occurred. This is because the source code at the time the walk occurs has already been passed over. This is not usually a grave inconvenience as editors and interactive development environments will locate the source line from information embedded in the error message.

Had the placement of the two functions been reversed as is shown in the module

```
    t2.cpp:

  1     int f( int n )
  2         { return  10 / n; }
  3     int g()
  4         { return f(0); }
```

then there would, by default, be no warning about the possible division by 0. This is because by the time we see the **`f(0)`** on line 4, the definition of **`f()`** is gone. It is for this reason that the multi-pass option was introduced. If we issue

```
lin -u -passes(2) t2.cpp
```

then the output we obtain is as shown below:

```
--- Module:   t2.cpp

/// Start of Pass 2 ///

--- Module:   t2.cpp

During Specific Walk:
  File t2.cpp line 4: f(0)
t2.cpp  2  Warning 414: Possible division by 0 [Reference:
    File t2.cpp: line 4]
```

The**-passes(2)** option requests two passes through the set of modules. Note that on some operating systems **-passes(2)** can't be used on the command line. For such systems, either **-passes=2** or**-passes[2]** should work. The output reveals, through a verbosity message, the start of pass 2 indicating that no messages were produced in the first pass. The message that we do get is otherwise identical to the earlier case.

We can, in some cases, deduce the return value or at least some properties of a return value for a specific function call. For example, given the module:

```
  t3.cpp:

1     int f( int n )
2         { return n - 1; }
3     int g( int n )
4         { return n / f(1); }
```

and the command

```
lin -u -passes(2) t3.cpp
```

we obtain the following output:

```
--- Module:   t3.cpp

/// Start of Pass 2 ///

--- Module:   t3.cpp
                     _
    { return n / f(1); }
```

```
t3.cpp  4  Warning 414: Possible division by 0 [Reference:
    File t3.cpp: lines 2, 4]
```

In pass 1 we learn that `f()` is called with an argument of 1. In pass 2, when we process function `f()`, we deduce that this argument will result in a return value of 0. Later in pass 2, when `g()` is processed, we report the division by 0. Note that the message is not preceded by the phrase "During Specific Walk". This is because the error was detected during normal general processing of the function `g()` without using a specific argument value for `g()`'s parameter.

Specific calls can generate additional calls and this process can repeat itself indefinitely if we have enough passes. Consider the following example:

```
  t4.cpp:

1     int f(int);
2     int g( int n )
3         { return f(2); }
4     int f( int n )
5         { return n / f(n - 1); }
```

Here a denominator of `f(n-1)` in line 5 is not viewed with suspicion until we realize that the call to `f(2)` results in a call to `f(1)`, which results in a call to `f(0)`, which forces the return value to be 0. Given the command line

```
      lin -u -passes(3) t4.cpp
```

we get as output:

```
--- Module:   t4.cpp
                    _
    { return f(2); }
t4.cpp  3  Info 715: Symbol 'n' (line 2) not referenced

/// Start of Pass 2 ///

--- Module:   t4.cpp

/// Start of Pass 3 ///

--- Module:   t4.cpp

During Specific Walk:
  File t4.cpp line 3: f(2)
  File t4.cpp line 5: f(1)
t4.cpp  5  Warning 414: Possible division by 0 [Reference:
    File t4.cpp: lines 3, 5]
```

Here it took a 3rd pass to determine a possible division by 0. To see why 3 passes were necessary see option: **`-specific_wlimit(`*`n`*`)`**.

Notice that the sequence of specific calls, **`f(2)`**, **`f(1)`**, is given as a prologue to the message itself.

11. SEMANTICS

11.2 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 11.1 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(`** ***`name`*** [**,** ***`sem`***] ... **)**

This associates the semantics ***`sem`*** ... with the named function ***`name`***. The semantics ***`sem`*** are defined below. If no ***`sem`*** is given, i.e. if only ***`name`*** 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.

11.2.1 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 identical to the semantic **S2** defined in Section 11.1.2 Function listing. As in Section 11.1.2 Function listing it is considered a Return semantic. See Section 11.1 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 defined in Section 11.1.2 Function listing as **S3**; it is the semantic used for the **exit()** function. This also is considered a Return semantic.

i*p** (e.g. **3p**) the ***ith 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 */
```

This semantic is identical to the **S1** semantic described in Section 11.1.2 Function listing.

initializer Some member functions are used to initialize members. They may be called from constructors or they may be called whenever the programmer wants to reset the state of a class to that which it would have immediately after construction. You may designate that a member function is an initializer using the **-sem** option. (The initializer semantic is a flag semantic). If a member is dubbed an initializer a complaint will be issued if it fails to initialize all of the data members. For example:

```
//lint -sem(A::init,initializer)
struct A
   {
   int n;
   int m;
   A();
   void init()
       { n = 0; }  // warning: m is not initialized
   };
```

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 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 not initialized
    }
```

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 (**.lnt** file).

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/FlexeLint 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 which 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.

`type(`*`i`*`)` indicates that the type of the *`i`*th argument is reflected back to become the type returned by the function. The built-in functions **`strchr`**, **`strrchr`**, **`strpbrk`** and **`strstr`** have all been pre-endowed with the **`type(1)`** semantic (in addition to other semantics they may have).

For example, the usual declaration of **`strchr()`** in C is:

```
char *strchr( const char *, int );
```

Since the return pointer points into the string buffer passed as first argument then a literal reading of the prototype could place **`const`** data in jeopardy. However, since **`strchr`** has been given the **`type(1)`** semantic, if the string buffer is **`const`**, the return pointer is considered **`const`** as well and the usual warnings will be issued on an attempt to assign this to a plain **`char *`** pointer.

In C++ this problem should not occur as **`strchr()`** is overloaded:

```
char *strchr( char *, int );
const char *strchr( const char *, int );
```

The **`type(`*`i`*`)`** semantic is an argument semantic and joins with other argument semantics. Thus the semantic specification for **`strchr()`** resembles:

```
-sem( strchr, r_null, 1p, type(1) )
```

This indicates that the first argument should be checked for NULL as well as having the **`type`** property. Were we to transfer the first argument semantics as:

```
-function( strchr(1), myfunc(2) )
```

then the second argument of **`myfunc()`** would have both properties.

`nulterm(`*`i`*`)` indicates that the *`i`*th argument is or will become nul-terminated. For example, a call to the built-in:

```
strcpy( a, b );
```

will allow us to presume that both arguments are nul-terminated. An array that contains a nul-termination or a pointer to such an array will have certain properties that we need to know about to avoid giving bogus messages. For example:

```
for( i = 0; i <= 10; i++ )
   {
   if( a[i] == 0 ) break;
   ...
   }
```

If array **`a`** is nul-terminated (such as a string constant) we won't get too upset if it fails to contain 11 characters.

`nulterm(`*`i`*`)` was designed for strings but it could also be used for pointer arrays as well.

`thread` designates that a function is the root of a thread. It is a flag semantic. This semantic and the following thread semantics are more fully described in the chapter on multi-threading. See **Chapter 12. Multi-Thread Support**.

`thread_mono` designates that the function is the root of a mono thread; i.e. the thread will have only one instance.

`no_thread` designates that a function is not a thread (used for designating that **`main`** is not a thread). It is a flag semantic.

`thread_lock` designates that the function will lock a mutex. It is a flag semantic.

`thread_unlock` designates that the function will unlock a mutex. It is a flag semantic.

`thread_protected` designates that the function is protected by a mutex (useful if, for some reason this is not deduced automatically). That is, only one thread at a time is presumed to execute the body of the function. It is a flag semantic.

`thread_safe` designates that the function is thread safe. This presumably overrides an option that may otherwise indicate that the function was **`thread_unsafe`**. It is a flag semantic.

`thread_unsafe` designates that the function is thread unsafe (Category #1). It is a flag semantic.

`thread_unsafe(`*`groupid`*`)` designates that the function is thread unsafe (Categroy 2). Two functions with the same ***`groupid`*** are considered as if they were manipulating the same static data. The groupid is the programmer's choice.

`thread_not[(`*`list`*`)]` where the optional argument list is a comma-separated ***`list`*** of thread names, designates that the function may not be called (to any call depth) by any of the listed threads. If no ***`list`*** is provided then no thread may call the function. **`thread_not`** is a flag semantic

`thread_only(`*`list`*`)` where the argument list is a comma-separated list of thread names, designates that the function may be called only by threads on the list. t**`hread_only`** is a flag semantic.

`thread_create(`*`i`*`)` designates that the function's ***`i`***th argument is a thread. It is an argument semantic

`exp` a semantic expression involving the expression elements described below:

`i`*`n`** denotes the ***`i`th argument, which must be integral (E.g. **`3n`** 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.

`i`*`p`** 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 ***`i`*`n`**, ***`i`*** may be `@` in which case the return value is indicated.

***`i`*`P`** is like ***`i`*`p`** 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` may be a macro (non-function type), enumerator, or **`const`** variable. The ***`identifier`*** is retained at option processing time and evaluated at the time of function call. If a macro, the macro must evaluate to an integral expression.

`malloc(` ***`exp`*** **`)`** attaches a **`malloc`** allocation flag to the expression. See the discussion of Return Semantics in Section 11.2.2 Semantic Expressions.

`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: **`?:`**

12. MULTI-THREAD SUPPORT

12.1 Overview

This facility allows one or more concurrently executing threads to be identified as being associated with particular root functions. A pair of (user-designated) functions can be associated with mutex locking and unlocking. Abnormalities in locking and unlocking are identified.

Starting with each root function, we deduce the set of static variables accessed by each thread and whether these accesses are or are not protected by mutex locks. By static variable, we mean any variable that has static storage duration. That is, any variable that is nominally static (whether within or outside a function) and variables that are considered global (i.e. **`extern`**) or, for C++, at namespace scope. Reports are made of unprotected access to static variables shared by one or more threads.

External functions can be identified as being thread unsafe individually or in groups. Access by multiple threads to such functions are reported if proper protections are not in place. Functions that are compatible with only a subset of threads can be identified.

12.2 Identifying Threads

Threads are almost always associated with a function and we will assume that to be the case here. That is, the static text associated with a thread consists of a function (designated as a thread root) and all functions called by that function to whatever depth. By default, **`main()`** is presumed to be a thread root, but options can overrule that presumption.

There are two basic ways of identifying threads: (1) via options and (2) automatically.

(1) Options to Identify Threads

The option:

```
-sem( function-name, thread )
```

will identify ***`function-name`*** as a thread. For example:

```
-sem( input_reader, thread )
```

will identify **`input_reader`** as a thread. Like all semantics, this option must be given before the function is declared or defined. The name of the function may be fully qualified if a member of a class or namespace.

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: "Thread ’f(void)’ has an unprotected write access to variable ’n’ which is used by thread ’f(void)’`**. 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`**. For example,

```
-sem( f, thread_mono )
```

will identify **`f`** as a mono thread. If **`f`** had been so identified in the example above, the diagnostic would not have been issued.

By default, **`main()`** would be regarded as a (mono) thread. Removing the thread property from a function is accomplished with the **`no_thread`** semantic.

```
            -sem( function-name, no_thread )
```

Example:

```
            -sem( main, no_thread )
```

(2) Automatic Identification of Threads

Using POSIX threads, we can automatically identify a thread because it is passed as the third argument to **`pthread_create`**. For example:

```
            ...
            pthread_create( &input_thread, NULL, do_input, NULL );
            ...
```

can indicate to us that **`do_input()`** is a separate thread.

We will assume 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).

If you are not using POSIX threads you will probably want to be able to transfer this **`pthread_create()`** property to another function of your choice.

Using function mimicry, we can write:

```
         -function( pthread_create(3), our_thread_create(2) )
```

and this will copy the properties of the 3rd parameter of **`pthread_create`** to a particular parameter of your choice. In the above we transfer the thread creation property to the 2nd parameter of **`our_thread_create`**.

We also provide (somewhat redundantly) a separate argument semantic to designate this property directly. The semantic

```
            thread_create(i)
```

identifies the *`i`*th argument of a function as being a thread endowing argument. Thus after:

```
            -sem( their_thread_create, thread_create(1) )
```

the call

```
            their_thread_create( reader, ... );
```

will identify **`reader`** as a thread.

12.3 Mutual Exclusion

To make a static analysis meaningful, we need to identify areas of the code in which only a single thread can operate. These are called critical sections or, as we will term them, ***`thread protected regions`***. The use or modification of global variables by two different threads is not considered a violation of thread safety, provided such access lies within protected regions.

There are two ways of identifying thread protected regions of code: (1) by option and (2) automatically.

(1) By option:

```
        -sem( function-name, thread_protected )
```

will identify ***`function-name`*** as a thread protected region of the program. Normally, this option would not be needed as it is expected that thread protected regions would be detected automatically. If the automatic methods fail, you can fall back on using this option. Just make sure that only one thread at a time will be permitted to use this function or, at least, those portions of the function that access global variables or make calls on functions that access global variables.

(2) Automatic detection:

An alternative to identifying a function as **`thread_protected`** is to identify which primitive functions can lock and unlock a mutex.

```
        -sem( function-name, thread_lock )
```

will identify ***`function-name`*** as a function that will lock a mutex.

```
        -sem( function-name, thread_unlock )
```

will identify ***`function-name`*** as a function that will unlock a mutex.

By default, functions **`pthread_mutex_lock()`** and **`pthread_mutex_unlock()`** have the **`thread_lock`** and **`thread_unlock`** properties respectively.

By identifying lock and unlock primitives, we have two advantages over the **`thread_protected`** function semantic. We have a finer grain control over the identification of areas of mutual exclusion. Also we can assist the user in finding mismatched lock and unlock primitives. For example:

```
     //lint -sem( lock, thread_lock )
```

```
//lint -sem( unlock, thread_unlock )

extern int g();
void lock(void), unlock(void);

void f()
    {
    lock();
    unlock();       // great
  //------------
    lock();
    if( g() )
        {
        unlock();
        return;     // ok
        }
    unlock();       // still ok
  //------------
    lock();
    if( g() )
        return;     // Warning 454
    unlock();
  //------------
    if( g() )
        {
        lock();
        unlock();
        unlock();   // Warning 455
        return;
        }
  //------------
    if( g() )
        lock();

    {               // Warning 456
    // do something interesting
    }
    if( g() )
        unlock();
    }               // Warning 454 and 456
```

In the example above, Warning **454** is issued if a return is encountered when a mutex is still locked. Warning **455** is issued if an unlock has been issued without a corresponding lock. A Warning **456** is issued if two execution paths are combined that do not have the same lock state. Even if an unlock is provided later under the same '`if`' condition the practice is considered unsafe.

12.4 Thread-Protected (TP) Regions

Any portion of a function between a mutex lock and its corresponding unlock is considered to be a thread-protected (TP) region. A common name for a TP region is a critical section.

A function identified as being **`thread_protected`** (using the **`-sem`** option) is considered to be TP and the entire function is said to be a TP region.

Any access (read or write) to a variable outside a TP region is considered non-protected. Any thread that can arrive at such a non-protected access, possibly through a sequence of calls in non-TP regions, is considered to have a non-protected access to the variable.

Consider the following example:

```
//lint -sem( reader, thread )
//lint -sem( lock, thread_lock )
//lint -sem( unlock, thread_unlock )

int x;
int y;
void create(...);
void lock(void);
void unlock(void);

void g(void);
void h(void);

void reader() { g(); }

void g(void)
    {
    lock();
    y = 1;
    h();
    unlock();
    }

void h(void) { x = y; }

int main()
    {
    create( reader );
    h();
    return 0;
    }
```

This example contains two threads: **`main()`** and **`reader()`** and some lapses in the protection of global variables. Function **`g()`** has a thread-protected region from which it calls **`h()`** and modifies **y**. The villain in the piece is thread **main()** which calls **h()** without a mutex lock. Messages issued are:

```
Warning 457: Thread ’main(void)’ has an unprotected write access to
    variable ’x’ which is used by thread ’reader(void)’
Warning 458: Thread ’main(void)’ has an unprotected read access to
    variable ’y’ which is modified by thread ’reader(void)’
```

12.5 Constructor-triggered mutex locking

Mutex locking and unlocking can be controlled by the constructor and destructor of a particular class. The previous example might be rendered as follows:

```
//lint -sem( reader, thread )
//lint -sem( Lock::Lock, thread_lock )
//lint -sem( Lock::~Lock, thread_unlock )

int x;
int y;
void create(...);
struct Lock
    {
    Lock();
    ~Lock();
    };

void g(void);
void h(void);

void reader() { g(); }

void g(void)
    {
    Lock lock;
    y = 1;
    h();
    }

void h(void) { x = y; }

int main()
    {
    create( reader );
    h();
    return 0;
```

```
        }
```

Here the destructor for **`Lock`** will be called automatically before the function is terminated. The analysis will be the same and the very same problems reported earlier are reported here.

12.6 Function Pointers

If a function has had its address taken, then we presume that we do not know the context of every call made upon 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.

Consider the tale of two functions, **`f1`** and **`f2`**, as presented in the example below.

```
//lint -sem( t, thread )
//lint -sem( f2, thread_protected )

int x;
int y = 0;
void s(...);
void g();

int f1() { return x; }
void f2() { y = x; }

void t()
    { g(); }

void h()
    { s( f1 ); s( f2 ); }
```

The fact that **`t`** is a thread is sufficient to trigger a thread analysis. Both **`f1`** and **`f2`** have their addresses taken and are subject to extended scrutiny. They both access static variables which would be cause for issuing warnings. But **`f2`** is identified as a **`thread_protected`** function and therefore is immune from this criticism. The one warning issued is

```
Warning 459: Function ’f1(void)’ whose address was taken has an
        unprotected access to variable ’x’
```

12.7 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 (i.e., a critical section).

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 catgory 3 if every call on 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 not 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. These have been identified in the literature as MT illegal (see **[31]** Table 12.1).

12.7.1 Thread Unsafe Functions (Category 1)

A function is considered thread unsafe (more commonly called ’MT unsafe’; see for example **[31]**, Chapter 12]) if it cannot be used concurrently from two different threads without the protection of a mutex lock. This is Category 1.

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.

You can identify functions that are thread unsafe by the semantic

```
-sem( function-name, thread_unsafe )
```

And although the need for this is not yet apparent, you may identify functions as thread safe by a similar semantic:

```
-sem( function-name, thread_safe )
```

This will be useful when you are using an option, yet to be discussed, that indicates that every function within a header is thread unsafe. See `+thread_unsafe_h` in **Section 12.7.4 Header Options**.

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.

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.

12.7.2 Category 2 Functions

An oft-occurring situation is when several external functions in a library (think `stdio.h`) 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. It might be argued that a category 1 function is just a category 2 function with a group of size 1. True, but when it comes to discussing options involving headers and directories the categorization will be useful.

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 semantic that specifies that a function is **`thread_unsafe`** is extended to include 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 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). Warnings are issued for each pair of functions from the **`xyz`** group that are called from different threads. For example, a warning is issued on the call to function **`z()`** because **`z()`** is called from a different thread but is in the same group as **`x()`** and **`y()`** and none of the calls are made from a protected region. There are also Informational messages (**`847`**) about the unprotected calls to thread unsafe functions.

12.7.3 Category 3 Functions

To detect abuses of Category 3 functions you need to enable Info **`847`** for that function. By default Info is already activated so it would seem that you would have very little to do.

But this means that an **`847`** would be issued whenever any function were called from an unprotected region. This is presumably not what you would like to have happen.

The solution is to use the option sequence:

```
            -e847 +esym( 847, cat3func )
```

where **`cat3func`** is, of course, one of the category 3 functions you want to detect.

Note that as of this writing there is no other way to designate a function as Category 3.

12.7.4 Header Options

Thread safety (or the lack thereof) of functions can also be designated by the header files that contain declarations for them. In a process analogous to the triplet of options **`libclass`**, **`libdir`** and **`libh`** to specify the set of headers that are library, there is a similar triplet of options to specify thread safety.

The option:

> **`+thread_unsafe_h(`** *header-name* **`)`**

specifies that all the functions declared in the given header (not otherwise specified with the **`-sem`** option) are thread unsafe. This is category 1 thread safety. That is, no grouping is implied.

By contrast

> **`+thread_unsafe_group_h(`** *header-name* **`)`**

specifies that all the functions declared in *header-name* are thread unsafe and also belong to the same group whose name is the same as *header-name*. That is, the functions become category 2 functions.

Multiple headers can appear in category 2 options such as, for example,

```
+thread_unsafe_group_h( stdio.h, stdlib.h )
```

This specifies that all the functions declared within either of the two headers belong to a group whose name is

```
"stdio.h,stdlib.h"
```

12.7.5 Directory Options

The option:

> **`+thread_unsafe_dir(`** *directory-name* **`)`**

specifies that all headers found in the named directory (unless otherwise specified) are category 1 headers. That is, all functions declared within the headers are category 1 functions and belong to no group.

This can be overridden in a number of ways. An option of the form

> **`-thread_unsafe_h(`** *header-name* **`)`**

indicates that the named header is not thread unsafe even if it were found in a *directory-name* of a **`+thread_unsafe_dir`** option. On the other hand, the option

> **`+thread_unsafe_group_h(`** *header-name* **`)`**

overrides the **`+thread_unsafe_dir`** option for the particular header making that header category 2.

The option

> **`+thread_unsafe_group_dir(`** *directory-name* **`[,`** *directory-name* **`] ... )`**

specifies that the files found in the directories are each category 2 and all are under the same group name formed by the obvious comma-separated concatenation of directory names.

By contrast:

> **`+thread_unsafe_group_dir(`** *directory-name* **`(*) )`**

note the trailing **`(*)`** -- will designate that each header found is a category 2 header with a group name equal to its own name.

12.7.6 Thread Unsafe Classifications

The most general option (but one capable of being overridden by the previous two) is of the form:

> **`+thread_unsafe_class(`** *identifier* **`[,`** *identifier* **`] ... )`**

The available identifiers are as follows:

> **`all`** -- All headers are thread unsafe. That is, all functions declared in headers are considered thread unsafe.
>
> **`ansi`** -- All standard C headers are thread unsafe (see **Section 6.1 Libraries Header Files** for a list).
>
> **`angle`** -- All headers specified by angle brackets are thread unsafe.
>
> **`foreign`** -- all headers that are found via a search list (**`-i`** option or **`INCLUDE`** environment variable) are thread unsafe.

In each case the degree of thread unsafety is category 1. By contrast, the option:

```
+thread_unsafe_group_class( identifier [, identifier ] ... )
```

designates category 2. The identifiers allowed are those listed above for the **`thread_unsafe_class`** option. The category 2 group name is the identifier itself with priority given to **`ansi`**, **`angle`**, **`foreign`** and **`all`** in that order. For example

```
+thread_unsafe_group_class( angle, all )
```

will place all headers included with angle brackets in the group named **`angle`** and will place all other headers in the **`all`** group.

12.7.7 Priorities in Thread Unsafety

To determine the thread safety of a particular function, **`f`**, the following steps are taken:

(1) If a semantic for **`f`** is given specifying either **`thread_safe`** or **`thread_unsafe`** with or without a group name, this determines whether or not **`f`** is thread safe and in which group it lies.

(2) Otherwise if **`f`** is declared in a header file and that header file had been designated by the **`+thread_unsafe_group_h`** option then **`f`** is **`thread_unsafe`** category 2 and the group name, if not previously given, is taken from the option.

(3) Otherwise if **`f`** is declared in a header file and that header file had been designated by the **`+thread_unsafe_h`** option, then **`f`** is **`thread_unsafe`** category 1.

(4) Otherwise if **`f`** is declared in a header file and that header file had been found in a directory designated by the **`+thread_unsafe_group_dir`** option, then **`f`** is **`thread_unsafe`** category 2 and the group name, if not previously given, is taken from the option.

(5) Otherwise if **`f`** is declared in a header file and that header file had been found in a directory designated by the **`+thread_unsafe_dir`** option, then **`f`** is **`thread_unsafe`** category 1.

(6) Otherwise if **`f`** is declared in a header file and that header file satisfies one of the classifications of the **`+thread_unsafe_group_class`** option then **`f`** is **`thread_unsafe`** category 2 and the group name is the most specialized of the classifications.

(7) Otherwise if **`f`** is declared in a header file and that header file satisfies one of the classifications of the **`+thread_unsafe_class`** option, then **`f`** is **`thread_unsafe`** category 1.

12.7.8 Category 4 Functions

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 ) )
      //lint -sem( c, thread_protected )

      void a();
      void b();
      void c()
          {
          b();
          }

      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()`**.

You would not normally use both the **`thread_not`** semantic and the **`thread_only`** semantic for the same function.

Note that the call upon **`b()`** is made from a protected region. Unlike Categories 1 through 3, protecting the call does not eliminate the message.

12.7.9 Category 5 Functions

Since a function in this category is not to be used at all you can employ the deprecate option (see **`-deprecate`**). For example, if function **`crash`** is not to be used at all use the option:

```
      -deprecate(function,crash,MT illegal)
```

Alternatively, and, perhaps, preferably, you may use the **`thread_not`** semantic. Thus

```
    -sem( crash, thread_not )
```

will have the effect of producing Warning **`462`** when **`crash`** is called. This has the benefit of providing you with the name of the thread that is invoking **`crash`**.

12.8 Thread Local Storage

Thread Local Storage (TLS) is storage that is allocated separately for each thread initiated. References to TLS data do not require locks or critical sections.

Two syntactic forms are recognized.

12.8.1 __thread

The reserved word **`__thread`** can be used as a modifier to identify a variable as having thread local storage (TLS). E.g.

```
            __thread int n;
```

identifies **`n`** as being thread local. The reserved word needs to be activated with:

```
            +rw(__thread)
```

12.8.2 __declspec(thread)

The declaration:

```
            __declspec(thread) int n;
```

will declare **`n`** to be thread local.

12.9 Atomic Access

12.9.1 Atomic Operations

An atomic operation on data is one which, once started, will not be interrupted (by the hardware) until completion. In this section we will describe how a programmer can designate that loads and stores of some types are atomic. It is the programmer's responsibility to know which types can be loaded and stored atomically.

For example, let `n` be a 64 bit integer emulated in software by two 32 bit integers. A load of the 64-bit integer may consist of a pair of machine instructions that each load 32 bits. Typically an interrupt can occur between these parts of the access and, hence, the 64-bit integer is not atomic.

Suppose, for example, we attempt to execute:

```
        x = n;
```

in thread A and that the read of `n` is indeed interrupted. Suppose further that another thread B is writing to `n`.   It would be possible to assign to `x` a value consisting of a portion of `n` before the write (by thread B) and a portion of `n` after the write. It is often the case that either value for `n`, either the one before the write by thread B or the one after the write by thread B would be perfectly OK. What is not OK would be this ill-formed combination. There is a movie called "The Fly" which provides an especially evocative analogy.

Any operation consisting of multiple machine instructions will generally be non-atomic. There are also operations consisting of a single machine instruction that are not atomic. For example, block move instructions are often designed to be non-atomic. In such cases, machine registers (that are saved and restored by interrupt processing) are used to hold the partial state of the move.

By default, we assume that values are read and written non-atomically. If a variable is read by thread A without a mutex lock and the same variable is written by thread B (even with a mutex lock), Warning **458** will be issued. However, if it is given that the variable can be read atomically, no warning need be issued. Since setting a mutex lock can be expensive (especially compared with a load) this can represent a significant speed improvement. By contrast, if threads A and B write to the same variable, even if the variable is considered atomic, a warning (**457**) will still be issued.

A variable is often not a value to be used in isolation but rather is correlated with other values. For example, the count of the number of entries of an array is correlated with the content of the array. Does it make sense to declare access to the count an atomic operation? Actually, yes. In a producer-consumer situation, the consumer may want to do a quick check to see if the count is not zero and, if so, enter a protected region where it can remove an item.

So far, so good; but suppose the consumer is accessing the count to compare it with the last one that it consumed. Here the count is not just a binary value (zero or not) but is a value to be compared. If the producer from its protected region is writing the count concurrently with the consumer reading the count, the count has to be written atomically. Otherwise the reader will be reading a value that is half old and half new with possible deleterious consequences.

Thus a variable declared to have a type that is deduced to be atomic must not only be read atomically, it must also be written atomically, if the variable is correlated with other values and is not just Boolean.

12.9.2 Atomic Types

The following option enables you to identify which types are considered atomic.

`-atomic(` *type-pattern* **`)`**

Any variable whose type matches the type-pattern will be considered atomic. An (atomic) *type-pattern* can be any of the following:

1) an unqualified scalar type; this will match the same scalar type with the possible addition of qualifiers.

    E.g. **`-atomic(int)`** matches **`int const`** but not **`int*`**.

2) the keyword **`any_type`**; this will match any type.

3) the keyword **`any_scalar`**; this will match any scalar.

4) a pointer to a type-pattern P; this will match a pointer of any qualification to a type T provided P matches T.

    E.g. **`-atomic(any_type *)`** matches **`int*`**, **`int**`** and **`int **const`** but not **`int`**.

5) a type-pattern qualified (on the right) by one of the keywords **`Atomic_`**, **`near`** or **`far`**. These are called the atomic modifiers. Systems which support **`near`** and **`far`** pointers are the exception rather than the rule. You shouldn't be using them to designate that a type is atomic unless your compiler supports them. **`Atomic_`** on the other hand is intended to be used even if your compiler does not recognize it. A pattern P modified by M will match a type T modified by M provided P matches T.

    E.g. **`-atomic(any_scalar Atomic_)`** matches **`int Atomic_`**, **`int *Atomic_`** and **`int Atomic_ *Atomic_`** but not **`int Atomic_ *`**.

Quick Notes:

a) A scalar type is any integral type, floating type, enum, or pointer.

b) The keywords **`any_type`** and **`any_scalar`** are keywords only for the duration of the **`-atomic`** option. They are never placed in the rest of your program.

c) The appearance of the **`-atomic`** option triggers the creation of the **`Atomic_`** keyword that serves as a type modifier. This can be used in your programs. For example, given:

```
-atomic( any_type Atomic_ )
```

then

```
float Atomic_ x;
```

indicates that **x** is atomic. Your compiler, however, will not recognize this modifier and so you will need something like the following somewhere in your code:

```
#ifndef _lint
#define Atomic_
#endif
```

d) As implied above, for a type to match a *type-pattern* it must have the same or more modifiers (in corresponding pointer levels) as the *type-pattern*. Again, the only modifiers considered are **near**, **far** and **Atomic_**.

For example:

```
//lint  -atomic( int * Atomic_ )
//lint  +rw( near )
// ...
int * Atomic_ p;                // atomic
int Atomic_ *q;                 // not atomic
int near Atomic_ *Atomic_ r;    // atomic
int Atomic_ *Atomic_ near s;    // atomic
int * Atomic_ const t;          // atomic
```

e) The **-atomic** option may appear in either an indirect file (a **.lnt** file) or in source code as part of a **lint** comment option. Indeed, for the option to refer to a type defined in the source code it is necessary to place the option in the source code at some point after the type's definition.

f) The argument to **-atomic** is actually processed twice. The first time is when the option is seen at which time the pattern is saved. The second time is at the start of processing source code at which time the pattern is compiled. This assuming the option is given before source code is processed. If the option is encountered in a lint comment the two steps are taken at that time. But in no case is a saved type-pattern recompiled.

g) Atomicity is not lost when the type goes through a typedef. E.g.

```
//lint -atomic( int Atomic_ )
typedef int Atomic_ Atomic;

Atomic x;         // variable x is an atomic int
```

h) The option **-atomic**(**any_type ***) indicates that every pointer is atomic

As a final example consider the following:

```
//lint -atomic( unsigned char )
unsigned char volatile ch;
unsigned char f()
    { return ch; }
void g()
    { ch = 0; }
```

In the above code, the **`-atomic`** option designates that the type **`unsigned char`** and all its qualified variants are atomic. Therefore, **`ch`** will be considered to be read atomically. This implies that in a multithreaded environment, where **`f()`** and **`g()`** can be called from two different threads, **`g()`** will have to be invoked from a thread protected region. But **`f()`** can be freely called by any thread at any time.

But suppose you don't want **`volatile`** characters to be considered atomic. Under these circumstances it is better to use the **`Atomic_`** qualifier.

```
//lint -atomic( unsigned char Atomic_ )
unsigned char volatile ch;        // not atomic
unsigned char Atomic_ atomic_ch;  // atomic
...
```

12.10 Declarative Methods

It is possible to use declarative mechanisms to guarantee coincidence of purpose between data (shared among multiple threads or not) and function (thread safe or not).

It so happens that there are a number of old modifier flags that can be used for this purpose. One such is the keyword '**`fortran`**'. This keyword is normally not active. When it was active, compilers would use the modifier as a clue to employ a fortran-like calling sequence for any function so designated. Lint would simply ignore these intended semantics and restrict its usage to ensuring that declarations would be consistent with respect to this modifier. For example, you wouldn't want to pass a pointer that points to a Fortran function to a pointer that points to a non-fortran function. These simple type-qualification semantics can be used as the basis for a new keyword, in this case '**`Shared`**'.

For example:

```
//lint -rw_asgn(Shared,fortran)
struct X { void f() Shared; void g(); ... };
X Shared a;
X b;
...
a.f();   // OK
```

```
a.g();   // Error
b.f();   // Error
b.g();   // OK
```

In this example **a** is shared among several threads. **x::f()** knows how to deal with the multiple thread situation. **X::g()** does not. So clearly **a.g()** is an error. On the other hand, **b** is not shared. Presumably it would be a mistake to invoke **b.f()** because the costly mechanism of mutex locking embedded within X**::f()** is not necessary.

Using functions rather than member functions we can obtain the same effect.

```
//lint -rw_asgn(Shared,fortran)
struct X { ... };
void f( struct X Shared * );
void g( struct X * );
struct X Shared a;
struct X b;
...
f( &a );   // OK
g( &a );   // Error
f( &b );   // Error
g( &b );   // OK
```

In addition to '**fortran**' there are a number of other old modifiers that could be employed including: '**pascal**', '**_fastcall**', and '**_loadds**'.

Any such modifier that is used in the formation of a type will be embedded within that type when the type is displayed for diagnostic purposes. The name that is used by default will be the original qualification name. This name will be overridden when the **-rw_asgn** option assigns a modifier to some new name.

13. OTHER FEATURES

13.12 MISRA Standards Checking

The Motor Industry Reliability Association (MIRA) released a programming guideline for C in 1998 (sometimes referred to as MISRA C1), and a revised version was released in 2004 (MISRA C2). In 2008, MIRA released guidelines for C++ (MISRA C++). PC-Lint/FlexeLint have supported checks for the available MISRA guidelines since early 2001, and we intend for Lint to provide ongoing and increasing support for these guidelines.

The primary way of activating MISRA checking for MISRA C2 guidelines is via the author file

```
au-misra2.lnt
```

This contains the appropriate options to activate and annotate Lint messages dealing with MISRA C2. To activate MISRA C1 checking, use the file **`au-misra1.lnt`**. To activate MISRA C++ checking, use the file **`au-misra-cpp.lnt`**.

Lint can report violations of several MISRA C rules with messages **`960`** and **`961`** and of C++ rules with messages **`1960`** and **`1963`**. Additional rules, are covered in other messages, the details of which you can find listed in the **`au-misra1.lnt`**, **`au-misra2.lnt`** and **`au-misra-cpp.lnt`** files.

To specify a particular MISRA C/C++ standard, you can use the **`-misra`** option. For example, if you want to specify MISRA C2 explicitly, you can use either: **`-misra(2)`**, **`-misra(C2)`** **`-misra(C2004)`**, but the preferred method is to use the **`au-misra2.lnt`** file.

Conversely, you can specify the 1998 MISRA C Standard by using either: **`-misra(1)`**, **`-misra(C1)`** or **`-misra(C1998)`**, but the preferred method is to use the **`au-misra1.lnt`** file.

Should MISRA release an additional C++ Standard, the **`-misra()`** option will possess the ability to specify the particular version.

Note: Although the descriptions for messages **`960`**, **`961`**, **`1960`**, and **`1963`** lists the MISRA rules covered by these messages, the best overall documentation on MISRA coverage is the appropriate **`au-misra`**... file.

13.13 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:

`&file=`*`filename`* This option designates the file to which the report will be written. This option must be present to obtain a report.

`&overhead(`*`n`*`)` establishes a call overhead of ***n*** bytes. The call overhead is the amount of stack consumed by a parameterless function that allocates no **`auto`** storage.

Thus if function **`A()`**, whose auto requirements are 10, calls function **`B()`**, whose auto requirements are also 10, and which calls no function, then the stack requirements of function **`A()`** are 20+***n*** where ***n*** is the call overhead. By default, the overhead is 8.

`&external(`*`n`*`)` 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.

`&summary` 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 **`974`** and is equivalent to issuing the option **`+e974`**. This option is not particularly useful since a summary report will automatically be given if a **`+stack`** option is given. It is provided for completeness.

`name`**`(`*****`n`*****`)`** where ***`name`*** 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. ***`name`*** may be a qualified name.

For 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`***.

(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.

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 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
’`%c`’ for the callee and
’`%e`’ for an ’`external`’ tag on the callee.

See `-format_stack` in Section 5.6.3 Message Format Option for more details. See also Message **974** in Section 19.6 C Elective Notes.

17. PROGRAM INFORMATION

PC-lint/FlexeLint can glean information about your program and present it in a form suitable for input into a data base or spreadsheet. This information can be used for many purposes. For example, it would be easy to use your favorite scripting language to write a short and simple program to read this data and then checks for conformance to naming conventions or other style rules or guidelines.

This option

`+program_info( output_prefix=``*Prefix*` [**,** *sub-option*] ... **)**

instructs Lint to output four text files that reflect some of its internal representation of your program. The information is separated into these four *categories*.

```
file
type
symbol
macro
```

For each category, Lint will output a file with a corresponding name.

```
Prefix file.txt
Prefix type.txt
Prefix symbol.txt
Prefix macro.txt
```

The data within each file are given by a list of *records* (one per line) composed of *fields*, which are the smallest units of representation. Example: in C++ programs, there is usually a global function called **`main`**. It will be represented as a record in **`symbol.txt`**. One of its data fields indicates that it is a function symbol (as opposed to, say, a variable symbol).

[Note: most information is expected to be output during Global Wrap-up time -- that is, after processing all project modules -- so whenever you make changes to the arguments to this option, you should first test your configuration against a single C or C++ source file before using it against your entire project.]

This option has two forms:

```
+program_info( options )

-program_info( options )
```

The second form is used to store away a sequence of options that you would use to display program information whenever you choose to do so. Presumably the option would be placed in a

.lnt file to be triggered with a simple **+program_info** on the command line. (See also **Section 17.3.)**

A required sub-option is **output_prefix=***Prefix*. *Prefix* may optionally be surrounded by double-quote characters so as to allow spaces and other non-alphanumeric characters in pathnames. *Prefix* need not indicate a directory; you may specify a simple filename prefix, a relative path (with or without a filename prefix), or the prefix of an absolute path. If you intend for *Prefix* to represent a directory, it should end with a trailing directory separator character (e.g. '/'). Example:

```
            +program_info( output_prefix="foo_" )
```

causes Lint to create the files **foo_file.txt**, **foo_type.txt**, **foo_symbol.txt** and **foo_macro.txt**. But:

```
            +program_info( output_prefix="foo_/" )
```

causes Lint to attempt to create **file.txt**, **type.txt**, **symbol.txt** and **macro.txt** within a directory called **foo_** (which is expected to exist already within the working directory). *Prefix* may be an empty string. E.g.:

```
            +program_info( output_prefix= )
```

results in an attempt to create output file streams for **file.txt**, etc., within the working directory.

Before proceeding, we recommend the reader try producing these files using some variation of the above options after first making sure that vital files that happen to have identical names will not be clobbered.

The generated information is organized in a way similar to that of a relational database. For example, for each record in **symbol.txt**, there is location information in the form of a file and line number. The file is given by a reference to exactly one record in **file.txt**. Similarly, the symbol's type is indicated by a reference to exactly one record in **type.txt**.

17.1 Record Fields

The following sections enumerate the set of fields available within each file.

17.1.1 The file category (*Prefix*file.txt)

INDEX	An integer that uniquely identifies a file.

`NAME`	The name of the file. The flag option **`'ffn'`** determines whether an absolute or relative path name is used

17.1.2 The `type` category (*`Prefix`*`type.txt`)

`INDEX` An integer that uniquely identifies a type.

`NAME` The name of the type (e.g., **`int *`**).

`KIND` The kind of type (e.g., "**`pointer`**"). Possible values for this field fall into two groups. There are primitive type kinds:

`bool, signed_plain_char, unsigned_plain_char, signed_char, unsigned char, signed_wchar_t, unsigned_wchar_t, signed_short, unsigned_short, signed int, unsigned_int, signed_long, unsigned_long, signed_long_long, unsigned_long_long, float, double, long_double,void`

and type kinds of more complicated types:

`pointer, bit_field, array, function, struct, union, enum, reference, pointer_to_member, member_function, member_data, template_parameter, namespace, dependent_specialization, dependently_dimensioned_array, dependent_name`

17.1.3 The `symbol` category (*`Prefix`*`symbol.txt`)

`FILE` A file index which refers to **`INDEX`** in **`file.txt`** and specifies the file in which the symbol is defined (or declared).

`LINE` An integer that indicates the line number within the specified file where the symbol is defined or declared.

The first line of a file is always regarded as being line 1 (as is the case in most text editor programs). A **`LINE`** value of zero indicates the symbol was created in the absence of a declaration in a source file. (E.g., **`namespace std`** is created implicitly so that **`std::bad_alloc`** and the implicitly declared allocation and deallocation operators may be created. So unless **`namespace std`** is seen, the symbol representing that namespace is said to be declared at line zero.

[Note: If the symbol was declared but not defined, then **`FILE`** and **`LINE`** refer to the location of the first declaration. If the symbol was defined, these fields refer to the definition.]

`NAME` The name of the symbol.

`LINKAGE` The symbol’s linkage. Possible values for this field are:

`no_linkage, internal_linkage, external_linkage`

(For authoritative descriptions of these terms, refer to the ISO standards.)

`SCOPE` The symbol’s scope. Possible values for this field are:

`function_scope, class_scope, namespace_scope, block_scope, file_scope`

(For authoritative descriptions of these terms, refer to the ISO standards.)

`KIND` The kind of symbol. E.g., enumeration constants are said to be of the kind "**`enumerator`**". For variables, this field indicates storage duration and will have a value of "**`auto`**" or "**`static`**". Possible values for this field are as follows:

`auto, static, function, member_function, instance_member, enumerator, type, label, class_template, function_template, namespace, template_parameter, using_declaration`

`TYPE` A type index which refers to **`INDEX`** in **`type.txt`** and specifies the type of the symbol.

17.1.4 The `macro` category (*`Prefix`*`macro.txt`)

`FILE` A file index which refers to **`INDEX`** in **`file.txt`** and specifies the file in which the macro is defined.

`LINE` An integer that indicates the line number within the specified file where the macro’s definition appears. As with the records in **`symbol.txt`**, the first line of a file is always regarded as being line 1. A **`LINE`** value of zero indicates the macro was created in the absence of a definition in a source file. (E.g., internally-defined macros and macros defined through a command-line option have a **`LINE`** value of zero.)

`NAME` The name of a preprocessor macro

`PARAMETER_COUNT` The number of parameters accepted by the macro. In the case of an object-like macro, this has a value of 0.

20. WHAT'S NEW

This chapter details the new and improved features of PC-lint/FlexeLint 9.00 over PC-lint/FlexeLint 8.00. In some rare cases an option was supported in 8.00 but documented only in the `readme.txt` file.

20.1 Major New Features

- Pre-compiled Headers

  Pre-compiled headers, as users of C and C++ systems are well aware, can dramatically reduce the time spent in processing multiple modules. See **Section 7.1 "Pre-compiled Headers"**.

- Static Variable Tracking

  We now incorporate variables of static storage duration in our value tracking. These include not only variables that are nominally static, as local to a function and local to a module, but also external variables.

  See **`-static_depth(`*`n`*`)`** and flag **`-fsv`**.

- Thread Analysis

  We examine multi-threaded programs for correct mutex locking and report on variables shared by multiple threads that are used outside of critical sections. See **Chapter 12. Multi-thread Support**

- Stack Usage

  We can report (Note **`974`**) on the overall stack requirements of any program whose function calls are non-recursive and deterministic (i.e. calls not made through function pointers). This is very useful for embedded systems development where the amount of stack required can be mission critical. A complete detailed report of stack usage for each function is available as well (See **`+stack()`** in **Section 5.7 Other Options**).

- Dimensional Analysis

  We now support through the strong type mechanism the classical dimensional analysis that engineers and physicists have traditionally employed in verifying equations. A ’type’ can now be a ratio or product of other types and the compound types are checked for consistency across assignment boundaries. See **Section 9.4 Multiplication and Division of Strong Types**.

- **`-deprecate`** option

The user may deprecate particular symbols in any of the following categories: **`function`**, **`keyword`**, **`macro`** and **`variable`**. See **`-deprecate`**.

- Message Enhancement and Control

  You may now enhance any message parameterized by ***`Symbol`*** so that the symbol's type is also given (See **`+typename`**).

  You may suppress any message parameterized by ***`Symbol`*** on the basis of the type of the symbol (See **`-etype`**).

  You can suppress messages parameterized by ***`String`*** on the basis of that string (See **`-estring`**).

  You may activate a message for a particular ***`Symbol`*** (or set of symbols) that is otherwise inhibited (see **`+esym`**).

  You may suppress a message while calling a particular function (see **`-ecall`**), while calling library functions (see **`-elibcall`**) and while invoking library macros (see **`-elibmacro`**).

- Enhanced MISRA Checking

  Continued improvements have been made to our suite of MISRA checks. These include detection of recursion, support for the MISRA 2 'underlying type' concept and determination of side effects for functions and MISRA C++. See the enabling files **`au-misra1.lnt`** for MISRA-C:1998, **`au-misra2.lnt`** for MISRA-C:2004 and **`au-misra-cpp.lnt`** for MISRA-C++: 2008.

- Source-echo mode

  Lint messages can optionally appear embedded within the context of the original source. See **`+source`**(***`sub-option`***).

- html support

  Output can appear in the html format, suitable for a browser and handsomely color coded. See **`+html`**(***`sub-option`***).

- Program Info

  A comprehensive collection of information about your program is optionally provided yielding information on files, types, symbols and macros for simple viewing or in a manner absorbable by a database or spreadsheet. This information can be used for many purposes, including naming-style conventions. See **`+program_info.`**

- Macro Scavenging

This feature turns PC-lint/FlexeLint into a seeker of built-in macros supported by a compiler and lying about within compiler header files. This is perfect for the unknown compiler with long and forgotten macros ready to trip up a third party processor such as PC-lint/FlexeLint. See **`-scavenge`**.

- New semantics

  A number of new semantics have been added to the **`-sem`** option:

`initializer`	indicates the member function can be relied upon to initialize all the members
`cleanup`	indicates that the function is expected to free or zero all pointer members.
`inout(`*`i`*`)`	indicates that the *`i`*th parameter will read as well as write to its (indirect) argument.
`pod(`*`i`*`)`	indicates that the *`i`*th argument requires a pointer to a POD (Plain Old Datatype).
`pure`	can be used to indicate that the function is a pure function.

  A number of new semantic flags support multi-threading analysis: See **`thread`**, **`thread_lock`**, **`thread_unlock`**, and **`thread_protected`** and many others.

New Messages

Version 9 has some 146 new messages. Some of the more prominent of these are as follows:

- Read-Write Analysis

  Ever wonder whether each assigned value to a (local) variable actually has a chance of being used before another value is assigned to the variable or before exiting the program? We now detect this condition (See messages**`438`** and **`838)`**.

  Appropriate options allow the programmer to customize this check to suit his or her programming style (See **`-fiw`** and **`-fiz`**).

- **`for`** clause Scrutiny

  `for` clauses are now subject to intense scrutiny. We complain if the variable tested in the 2nd expression is not the same as the variable modified in the third or the variable initialized in the first. (Warnings **`440`**, **`441`** and **`443`**). We warn if the testing direction (2nd expression) seems inconsistent with the increment direction (3rd expression), (Warning **`442`**), or if the expression tested is inconsistent with the expression incremented (Warning **`444`**) or if the loop index variable is modified in the body of the loop (Warning **`850`**).

- Pre-determined Predicates

We can detect in a variety of circumstances that the value of a predicate is pre-determined to be true or false. For example:

```
unsigned u; ...
    ( u & 0x10 ) == 0x11
```

is always false (**587**) or

```
unsigned u; ...
   ( u = 2 ) != 1
```

has a predictable outcome barring overflow (**588**), or

```
int n; ...
    ( n % 2 ) == 2
```

will always be false making the usual assumptions about integer division (**589**), or

```
int n; ...
    ( 5 << n ) < 0
```

will not be true unless there is a standards violation (**590**).

- Constants

  Constants come under careful examination. Within string or character constants we look for the psuedo-hex character `\0xFF` (message **693**), decimal characters following an octal escape sequence (**692**), or the embedded nul (**840**). We also look for numeric constants that have different types depending on language dialect (**694**).

- Expressions

  We report on compile-time zero's being added, multiplied, ORed, etc. to expressions (**835**) and on deducable zero's added, multiplied, etc. (**845**). Our order-of-evaluation checking has been extended to include the case of functions modifying objects (**864**) and the suspicious looking: `p ? a : b = 1` (message **1563**). We also look for non 0/1 assignments to boolean typed objects (**1564**). We issue warnings about unusual `sizeof` arguments (**866**).

- `const` qualification

  We now report on global or static variables that could be made `const` (**843**) and global or static pointers that could be declared as pointing to `const` (**844**).

- Unusual Declarations

We report on the following declaration

```
int a:5;
```

as its meaning is not defined by the standard (**`846`**). We also look for assignment operators that do not return a reference to a `const` ref (**`1941`**).

- Initializer Functions

  A new feature of Version 9 is the notion of an initializer. See the `initializer` semantic. Message **`1565`** warns when an initializer fails to carry out its mission and message**`1566`** points out when an initializer may be needed. There are similar considerations for the `cleanup` semantic.

- Include Guards

  Headers are examined to determine whether they contain standard `include` guards (**`451`** and **`967`**).

- Pointer and Reference Anomalies

  New messages that involve the ever dangerous pointer and the innocent appearing reference are as follows. We look for

  pointers to `auto` being assigned to a member of the current `class` (**`1414`**)
  pointers to non-POD `class` being passed to POD-seeking functions such as `memcpy` (**`1415`**)
  string constants assigned to initialize non-`const char` pointers (**`1778`**)
  an uninitialized reference used to initialize another reference (**`1416`**)
  a reference member not appearing in a mem-initializer (**`1417`**)
  returning the address of a reference parameter (**`1780`**)
  passing the address of a reference parameter into the caller's address space (**`1781`**)
  assigning the address of a reference parameter to a static variable (**`1782`**, **`1940`**).

- Unreferenced (but constructed) class variables

  Ever try to find all those unreferenced `CString`'s? We didn't complain in the past because technically speaking, they were referenced by their own constructor. Such variables now get a new Info message (**`1788`**) so they can be picked out and eliminated.

20.2 New Error Inhibition Options

See Section 5.2 Error Inhibition Options

`-/+ecall(`*`#,Name1`*[, *`Name2`* ...]) inhibits (**`-`**) or re-enables (**`+`**)
error message # based on the name of some function (or some wildcard name pattern) while

a call to function ***`Name1`*** (or ***`Name2`***, etc.) is being parsed.

`-/+elibcall(``#[,#]`**`)`** inhibits (**`-`**) or re-enables(**`+`**) the numbered messages while parsing calls to library functions.

`-elibmacro(``#`**`)`** is like **`-emacro(`**`#,`***`Name`*****`)`** except that it applies to all macros defined in library code.

`+/-etype(``#,`***`Name1`***`[,` ***`Name2`***`, ...]`**`)`** enables (+) or suppresses (-) messages numerically equal to or matching `#` which are parameterized by at least one symbol whose type is identical to or which matches one of ***`Name1`***, ***`Name2`***, ...

`+efreeze`	inhibits subsequent error suppression options
`-efreeze`	reverses the effect of **`+efreeze`**
`++efreeze`	locks in freezing permanently

`-/+efreeze(``w#` `[,w#` ... `]`**`)`** Behaves like **`-/+efreeze`**, but acts only on messages in the given warning level(s).

`+esym(``#,`***`name`*****`)`** can be used to override a **`-e`**`#` just as a **`-esym`** can override a **`+e`**`#`.

`-/+estring(``#,`***`String1`***`[,` ***`String2`***`, ...]`**`)`** inhibits (**`-`**) or re-enables (**`+`**) messages based on strings used as Lint message parameters such as ***`Context`***, ***`Kind`***, ***`Location`***, ***`String`***, ***`Type`***, and ***`TypeDiff`***.

20.3 New Verbosity Options

See Section 5.4 Verbosity Options

`-va`... will cause a message to be printed each time there is an Attempt to open a file.

`-ve`... will cause a message to be printed each time a template function is instantiated.

20.4 New Flag Options

See Section 5.5 Flag Options

`f@m`	commercial `@` is a Modifier flag
`fat`	Parse **`.net`** ATtributes flag
`fda`	Double-quotes to Angle brackets flag
`fdd`	Dimension by Default flag
`fdh`	dot-h flag - Enhanced
`fet`	Explicit Throw flag
`ffc`	Function takes Custody flag

`fii`	Inhibit Inference flag
`fiw`	Initialization-is-considered-a-Write flag
`fiz`	Initialization-by-Zero-is-considered-a-Write flag
`fjm`	JM controls the Multiplier group flag
`fld`	Label Designator flag
`fmc`	Macro Concatenation Flag
`fns`	Nested Struct flag
`fnr`	Null-can-be-Returned flag
`fpd`	Pointer-sizes Differ flag
`fqb`	Qualifiers-Before-types flag
`frn`	treat carriage Return as Newline
`fsg`	Std is Global flag
`fsn`	Strings as Names flag
`fsv`	track Static Variable flag
`fus`	Using namespace Std flag
`fvl`	Variable Length array flag
`fwm`	**`wprintf`** formatting follows Microsoft flag

20.5 New Message Presentation Options

See Section 5.6.1 Message Height Options

`-hS`... will force a line skip in both places.

`-he`... places the source line (and the indicator and the macro expansion) at the end of the message.

See Section 5.6.3 Message Format Option

`-format_stack=`... controls the detailed formatting of the output produced by **`+stack`**.

`-format_template=`... controls the detailed formatting of a prologue to any message that is issued while instantiating a class template.

`-format_verbosity=`... controls the detailed formatting of the verbosity output when the **`+html`** option is used. Its primary purpose is to allow the user to add font information to the verbosity.

20.6 Additional Other Options

See Section 5.7 Other Options

-A(***Language Year***) This option allows you to specify the version you are using with the **-A** option.

-align_max(*option*) This option (patterned after the Microsoft **pragma packed**) allows the programmer to temporarily set the maximum alignment of any data object.

++b Use **++b** to place the banner line onto **-os(*file*)**

+/-compiler(*flag1* [,***flag2*** ...]**)** This option allows the programmer to specify flags that describe compiler-specific behavior.

-deprecate(***category***, ***name***, ***commentary*** **)** deprecates use of a ***name.*** You may indicate that a particular name is not to be employed in your programs.

-d*name*{*definition*} This is an alternative to **-d*name*=*definition***. **-d*name*{*definition*}** has the advantage that blanks may be embedded in the definition.

+html(*sub-option*,...) is used when the output is to be read by an HTML browser.

-indirect(***options-file*** **[**,...**]** allows you to specify Lint option files to be processed when this option is encountered.

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

+libm(***module-name*** **[**,...**]** allows you to specify modules to be treated as library files.

-maxfiles(*n*) A preset limit on the maximum number of files is approximately 6,400. This limit can be changed by specifying a new limit with the **-maxfiles** option.

-message(*text*) will allow the user to issue a special Lint message that will print '***text***' only at the time that this option is encountered.

-restore can be used on the command line or within a "**.lnt**" file.

-restore_at_end using this option has the effect of surrounding each source filename argument with **-save** and **-restore**.

-save can be used on the command line or within a "**.lnt**" file.

-scavenge(***filename-pattern*** [, ...] **| clean**, ***filename***)
this option automatically finds compilers' built-in macros. It completely changes the character of Lint from static analyzer to a scavenger of macros (or cleanup facility depending on the sub option).

-setenv(*directive*) will allow the user to set an environment string.

+source(*sub-option*,...) will cause all source lines of a file or files to be echoed to the output stream.
-source(*sub-option*,...) will enable the user to specify sub-options without triggering the echoing.

+stack(*sub-option*,...) The **+stack** option can be used to generate a report on cumulative stack requirements.

-static_depth(*n*) adjusts the depth of static variable analysis.

-strong(*flags*, *name*, ...) can be used to specify the ***name*** of one of the built-in types: **bool**, **char**, **signed char**, **unsigned char**, **wchar_t**, **short**,**unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **long long**, **unsigned long long**, **float**, **double**, **long double**, **void**.

-subfile(*indirect-file*, options|modules)
This is an unusual option and is meant for front-ends trying to achieve some special effect.

-summary causes a summary of all issued messages to be output after global wrap-up processing.

-tr_limit(*n*) allows the user to specify a template recursion limit.

+typename(*#*[,*#* ...]) For each message equal to or matching *#*, **+typename(*#*)** will cause PC-lint/FlexeLint to add type information for any and all symbol parameters cited in the message.

+xml(*name*) By adroit use of the **-format** option you may format output messages in **xml**. This option has two purposes. Special **xml** characters ('**<**', '**>**' and '**&**' at this writing) will be escaped (to "**&lt;**", "**&gt;**" and "**&amp;**" respectively) when they appear in the variable portion of the format. Secondly, if ***name*** is not null, the entire output will be bracketed with **<*name*>** ... **</*name*>**. If ***name*** is null this bracketing will not appear.

20.7 Compiler Adaptation

See Section 11.1.1 Special Functions

___**assert** In addition to the __**assert()** function (two underscores) there is now the ___**assert()** function (three underscores). The latter differs from the former in that it always returns.

See Section 5.8.12 Additional Keywords

`__packed` A **`struct`** (or **`class`**) may be declared as **`__packed`** (two leading underscores) to indicate that alignment is ignored when allocating space for data members of the **`struct`**.

See Section 5.8.3 Customization Facilities

`__typeof__` is similar in spirit to **`sizeof`** except it returns the type of its expression rather than its size.

`_up_to_brackets` is a potential reserved word that will cause it and all tokens up to and including the next bracketed expression to be ignored.

20.8 New Messages

See Section 19.1 C Syntax Errors

```
95   Expected a macro parameter but instead found 'Name'
98   Recovery Error (String)
109  The combination ’short long’ is not standard, ’long’is assumed
120  Initialization without braces of dataless type ’Symbol’
121  Attempting to initialize an object of undefined type ’Symbol’
143  Erroneous option: String
158  Assignment to variable ’Symbol’ (Location) increases capability
159  enum following a type is non-standard
160  The sequence ’( {’ is non standard and is taken to introduce a GNU
     statement expression
161  Repeated use of parameter ’Symbol’ in parameter list
```

See Section 19.3 Fatal Errors

```
317  File encoding, String, not currently supported; unable to continue
327  Bad pipe, code Integer
328  Bypass header 'Name' follows a different header sequence than in
     module 'String' which includes File1 where the current module
     includes File2
```

See Section 19.4 C Warning Messages

```
431  Missing identifier for template parameter number Integer
438  Last value assigned to variable 'Symbol' not used
440  for clause irregularity: variable 'Symbol' tested in 2nd
     expression does not match 'Symbol' modified in 3rd
441  for clause irregularity: loop variable 'Symbol' not found in 2nd
     for expression
442  for clause irregularity: testing direction inconsistent with
     increment direction
```

443 for clause irregularity: variable '*Symbol*' initialized in 1st expression does not match '*Symbol*' modified in 3rd

444 for clause irregularity: pointer '*Symbol*' incremented in 3rd expression is tested for NULL in 2nd expression

445 reuse of for loop variable '*Symbol*' at '*Location*' could cause chaos

447 Extraneous whitespace ignored in include directive for file ’*FileName*’; opening file ’*FileName*’

448 Likely access of pointer pointing *Integer* bytes past nul character by operator '*String*'

451 Header file '*FileName*' repeatedly included but does not have a standard include guard

453 Function ’*Symbol*’, previously designated pure, *String* ’*Name*’

454 A thread mutex has been locked but not unlocked

455 A thread mutex that had not been locked is being unlocked

456 Two execution paths are being combined with different mutex lock states

457 Thread ’*Symbol1*’ has an unprotected write access to variable ’*Symbol2*’ which is used by thread ’*Symbol3*’

458 Thread ’*Symbol1*’ has an unprotected read access to variable ’*Symbol2*’ which is modified by thread ’*Symbol3*’

459 Function ’*Symbol*’ whose address was taken has an unprotected access to variable ’*Symbol*’

460 Thread ’*Symbol*’ has unprotected call to thread unsafe function ’*Symbol*’ which is also called by thread ’*Symbol*’

461 Thread ’*Symbol*’ has unprotected call to function ’*Symbol*’ of group ’*Name*’ while thread ’*Symbol*’ calls function ’*Symbol*’ of the same group

462 Thread ’*Symbol*’ calling function ’*Symbol*’ is inconsistent with the ’*String*’ semantic

464 Buffer argument will be copied into itself

522 Highest operator or function lacks side-effects

583 Comparing type ’*Type*’ with EOF

585 The sequence (??*Char*) is not a valid Trigraph sequence

586 String ’*Name*’ is deprecated. *String*

587 Predicate ’*String*’ can be pre-determined and always evaluates to *String*

588 Predicate ’*String*’ will always evaluate to *String* unless an overflow occurs

589 Predicate ’*String*’ will always evaluate to *String* assuming standard division semantics

590 Predicate ’*String*’ will always evaluate to *String* assuming standard shift semantics

591 Variable ’*Symbol*’ depends on the order of evaluation; it is used/modified through function ’*Symbol*’ via calls: *String*

592 Non-literal format specifier used without arguments

593 Custodial pointer ’*Symbol*’ (*Location*) possibly not freed or returned

687 Suspicious use of comma operator

688 Cast used within a preprocessor conditional statement
689 Apparent end of comment ignored
690 Possible access of pointer pointing *Integer* bytes past nul character by operator ’*String*’
691 Suspicious use of backslash
692 Decimal character ’*Char*’ follows octal escape sequence ’*String*’
693 Hexadecimal digit ’*Char*’ immediately after ’*String*’ is suspicious in string literal.
694 The type of constant ’*String*’ (precision *Integer*) is dialect dependent
695 Inline function ’*Symbol*’ defined without a storage-class specifier ('static' recommended)
696 Variable ’*Symbol*’ has value ’*String*’ that is out of range for operator ’*String*’
697 Quasi-boolean values should be equality-compared only with 0
698 Casual use of realloc can create a memory leak

See Section 19.5 C Informational Messages

705 argument no. *Integer* nominally inconsistent with format
706 (argument no. *Integer*) indirect object inconsistent with format
707 Mixing narrow and wide string literals in concatenation
835 A zero has been given as [left/right] argument to operator ’*Name*’
836 Conceivable access of pointer pointing *Integer* bytes past nul character by operator ’*String*’
838 Previously assigned value to variable ’*Symbol*’ has not been used
839 Storage class of symbol ’*Symbol*' assumed static (*Location*)
840 Use of nul character in a string literal
843 Variable ’*Symbol*’ (*Location*) could be declared as const
844 Pointer variable ’*Symbol*’ (*Location*) could be declared as pointing to const
845 The [left/right] argument to operator ’*Name*’ is certain to be 0
846 Signedness of bit-field is implementation defined
847 Thread ’*Symbol*’ has unprotected call to thread unsafe function ’*Symbol*’
849 Symbol ’*Symbol*’ has same enumerator value ’*String*’ as enumerator ’*Symbol*’
850 for loop index variable ’*Symbol*’ whose type category is ’*String*’ modified in body of the for loop
864 Expression involving variable ’*Symbol*’ possibly depends on order of evaluation
866 Unusual use of ’*String*’ in argument to sizeof

See Section 19.6 C Elective Notes

904 Return statement before end of function ’*Symbol*’
905 Non-literal format specifier used (with arguments)
948 Operator ’*String*’ always evaluates to [True/False]

962 Macro ’*Symbol*’ defined identically at another location (*Location*)
963 Qualifier const or volatile follows/precedes a type; use -fqb/+fqb to reverse the test
967 Header file ’*FileName*’ does not have a standard include guard
974 Worst case function for stack usage: *String*
975 Unrecognized pragma ’*Name*’ will be ignored

See Section 19.7 C++ Syntax Errors

1020 template specialization for '*Symbol*' declared without a ’template<>’ prefix
1081 Object parameter does not contain the address of a variable
1082 Object parameter for a reference type should be an external symbol
1086 Compound literals may only be used in C99 programs
1087 Previous declaration of ’*Name*’ (*Location*) is incompatible with ’*Name*’ (*Location*) which was introduced by the current using-declaration
1088 A using-declaration must name a qualified-id
1089 A using-declaration must not name a namespace
1090 A using-declaration must not name a template-id
1091 ’*Name*’ is not a base class of ’*Name*’
1092 A using-declaration that names a class member must be a member-declaration
1093 A pure specifier was given for function ’*Symbol*’ which was not declared virtual
1094 Could not find ’)’ or ’,’ to terminate default function argument at *Location*
1095 Effective type ’*Type*’ of non-type template parameter #*Integer* (corresponding to argument expression ’*String*’) depends on an unspecialized parameter of this partial specialization

See Section 19.9 C++ Warning Messages

1405 Header <typeinfo> must be included before typeid is used
1414 Assigning address of auto variable ’*Symbol*’ to member of this
1415 Pointer to non-POD class ’*Name*’ passed to function ’*Symbol*’ (*Context*)
1416 An uninitialized reference ’*Symbol*’ is being used to initialize reference ’*Symbol*’
1417 reference member ’*Symbol*’ not initialized by constructor initializer list
1558 ’virtual’ coupled with ’inline’ is an unusual combination
1562 Exception specification for ’*Symbol*’ is not a subset of ’*Symbol*’ (*Location*)
1563 Suspicious third argument to ?: operator
1564 Assigning a non-zero-one constant to a bool
1565 member ’*Symbol*’ (*Location*) not assigned by initializer function

1566 **member ’*Symbol*’ (*Location*) might have been initialized by a separate function but no ’-sem(*Name*,initializer)’ was seen**
1567 **Initialization of variable ’*Symbol*’ (*Location*) is indeterminate as it uses variable ’*Symbol*’ through calls: ’*String*’**
1568 **Variable ’*Symbol*’ (*Location*) accesses variable ’*Symbol*’ before the latter is initialized through calls: ’*String*’**
1569 **Initializing a reference with a temporary**
1570 **Initializing a reference class member with an auto variable ’*Symbol*’**
1571 **Returning an auto variable ’*Symbol*’ via a reference type**
1572 **Initializing a static reference variable with an auto variable ’*Symbol*’**
1573 **Generic function template ’*Symbol*’ declared in namespace associated with type ’*Symbol*’ (*Location*)**
1576 **Specialization of template ’*Symbol*’ not declared in same file as primary template**
1577 **Partial or explicit specialization does not occur in the same file as primary template ’*Symbol*’ (*Location*)**
1578 **Pointer member ’*Symbol*’ (*Location*) neither freed nor zeroed by cleanup function**
1579 **Pointer member ’*Symbol*’ (*Location*) might have been freed by a separate function but no ’-sem(*Name*,cleanup)’ was seen**

See Section 19.10 C++ Informational Messages

1713 **Parentheses have inconsistent interpretation**
1777 **Template recursion limit (*Integer*) reached, use -tr_limit(n)**
1778 **Assignment of string literal to variable ’*Symbol*’ (*Location*) is not const safe**
1780 **Returning address of reference parameter ’*Symbol*’**
1781 **Passing address of reference parameter ’*Symbol*’ into caller address space**
1782 **Assigning address of reference parameter ’*Symbol*’ to a static variable**
1784 **Symbol ’*Symbol*’ previously declared as "C", compare with *Location***
1785 **Implicit conversion from Boolean (*Context*) (*Type* to *Type*)**
1786 **Implicit conversion to Boolean (*Context*) (*Type* to *Type*)**
1787 **Access declarations are deprecated in favor of using declarations**
1788 **Variable ’*Symbol*’ (*Location*) (type ’*Name*’) is referenced only by its constructor or destructor**
1789 **Template constructor '*Symbol*' cannot be a copy constructor**
1790 **Base class ’*Symbol*’ has no non-destructor virtual functions**
1791 **No token on this line follows the 'return' keyword**
1793 **While calling ’*Symbol*’: Initializing the implicit object parameter ’*Type*’ (a non-const reference) with a non-lvalue**
1794 **Using-declaration introduces ’*Name*’ (*Location*), which has the same parameter list as ’*Name*’ (*Location*), which was also introduced here by previous using-declaration ’*Name*’ (*Location*)**

1795 Defined template '*Symbol*' was not instantiated

1796 Explicit specialization of overloaded function template '*Symbol*'

1917 Empty prototype for *String*, assumed '(void)'

1940 Address of reference parameter '*Symbol*' transferred outside of function

1941 Assignment operator for class '*Symbol*' does not return a const reference to class

1942 Unqualified name '*Symbol*' subject to misinterpretation owing to dependent base class

1960 Violates MISRA C++ Required Rule *Name*, *String*

1963 Violates MISRA C++ Required Rule *Name*, *String*