Reference Manual

for

PC-lint/FlexeLint

A Diagnostic Facility

for

C and C++

Software Version 9.00 and Later
Document Version 9.00

September, 2008

Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426
(610)584-4261 (Voice)
(610)584-4266 (fax)

www.gimpel.com

Disclaimer

Gimpel Software has taken due care in preparing this manual and the programs and data on the electronic disk media (if any) accompanying this book including research, development and testing to ascertain their effectiveness.

Gimpel Software makes no warranties as to the contents of this manual and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Gimpel Software further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

Trademarks

PC-lint and FlexeLint are trademarks of Gimpel Software. All other products and brand names used in this manual are registered trademarks or tradenames of their respective holders.

1. BASIC FACTS

The software described in this manual is distributed in one of two ways. For the PC market (Microsoft Windows) the product is distributed in binary executable format. For all other platforms, it is distributed in shrouded C source code format and is known as FlexeLint. We use the term PC-lint/FlexeLint throughout this manual as a generic term to identify behavior that is common to both systems.

PC-lint/FlexeLint is a software package that finds errata in C and C++ programs consisting of one or more modules. It uses C99 **[4]** (or optionally C89) as the assumed standard for C with historical allowances for K&R C **[1]**. It uses ISO C++ 2003 **[34]** (or optionally C++98 **[10]** as the standard for C++.

Total memory requirements will depend on the size of user programs. In particular, the storage requirements to a first approximation depend upon the size of header files. A crude estimate of storage usage is 1Mb of additional memory needed for each 2000 lines of header files.

The FlexeLint package is distributed as "shrouded" C source code and may be compiled for any system. Details concerning such installation can be found in the "FlexeLint Installation Guide" (AKA "FlexeLint Installation Notes") that is provided with the FlexeLint package.

1.1 About the Document

Our on-line document (which you are reading now) also serves as on-line help. Although it requires an Adobe Acrobat Reader (TM), it has the great advantage of portability across many systems.

A perfect bound printed manual is also available. See our website for pricing and ordering information.

2. INTRODUCTION

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

2.1 An Example

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

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

As far as most compilers are concerned, it is a valid C (or C++) program. However, it has a number of subtle errors and question marks that will be reported upon by PC-lint/FlexeLint.

The string assigned to **`name[11]`** in line 8 drops the **`nul`** character; the comparison in line 11 is flawed; variable **`k`** in line 13 had not yet been initialized; variable **`kk`** in line 15 is conceivably not initialized; variable **`result`** in line 17 is possibly uninitialized; and the address of an **`auto`** is being returned in line 18.

Most C programmers can readily identify such irregularities in small programs. However, picking out such flaws in programs tens and hundreds of thousands of lines long is a job more appropriate to machine than man ... or madam.

2.2 A Lint for C++

C++ is an extraordinary language. No less a computer authority than Ray Duncan has called the language "one of the most grotesque and cryptic languages ever created".

Nonetheless, if C++ did not exist it would have to be created. Whereas C is a remarkably powerful systems programming language, it does not have complex numbers like Fortran, or strings like Basic, or subscript bounds checking like Pascal or the Objects of Smalltalk. With C++, you do not have these things either but you have something more important ... the ability to create these and many other programming constructs that a particular problem domain may require. With all this power there is bound to be a mixture of confusion, reckless hype, sorrowful disappointment, exaggerated claims, total rejection, and extreme euphoria.

This writer is confident that, in time, the language and its capabilities will be better understood and that its snares and pitfalls will be more readily identified. It is our hope that PC-lint/FlexeLint can make its contribution toward these ends, particularly the latter.

The original raison d'etre for lint was the relatively loose type checking of the original K&R C language and the multiple module (independently compiled) approach to large C programming. Lint filled these two needs by providing both stricter type checking within modules and cross-checking among several modules. ANSI/ISO C, to some extent, and C++, to a much greater extent, offer stricter type checking. Pointers can no longer masquerade as **`int`** or as anything other than as a pointer to the type to which it points; **`enum`** cannot freely be mixed with **`int`**; functions require a prototype and through name mangling, the prototypes must agree across modules. Even the type-loose **`printf()`** and **`scanf()`** can be replaced by their type-safe **`iostream.h`** equivalents **`cout`** and **`cin`**. What is there left for a lint to do?

While compiler technology has evolved to insure against the kinds of errors that lints would find 15 years ago, lint tools also have evolved. To take a very crude measure, the 27 warnings and 9 informational messages of the original PC-lint in 1985 have grown to 320 warnings and 223 informational messages in its current release. While several of these messages are no longer relevant to C++ coding, most still are. For example, expression analysis includes a lookout for unusual combinations of operators, order of evaluation problems, loss of precision, signed/ unsigned mismatches, unusual constants, suspicious comparisons, unusual indentation, suspicious truncation, unintended name hiding, suspicious initialization, inappropriate use of pointers to auto variables and macro irregularities.

In spite of C++'s many sophisticated features, it is still possible to create buggy programs. The type-checking in C++ is still not as strict as in, say, Pascal but by using PC-lint/FlexeLint's strong type option (See Chapter 9. STRONG TYPES.) type checking can be made arbitrarily strict. Uninitialized variables can still be a problem with C++. Our original simple scan for uninitialized

variables has been replaced by a sophisticated flow-of-control analysis aimed at uncovering pathways that could lead to a variable being used before initialization. See Section 10.1 Initialization Tracking

With version 7.00 we introduced, for the first time in any lint, inter-statement value tracking. By this, variable values and member values can be remembered or deduced. If later, such values are inappropriate for a particular context so that, for example, they could cause an out-of-range subscript or an access of a NULL pointer, an appropriate message can be given. See Section 10.2 Value Tracking As an enabling technology, value tracking can be used to check the calls to a large number of standard library functions. In addition, these checks can be transferred to user functions. See Section 11.1 Function Mimicry (-function)

PC-lint/FlexeLint can look across multiple modules. Even if all functions have been prototyped, declaration mismatches of function and data can still occur across modules and this can lead to mysterious program behavior. Global data may not have been initialized or not accessed; prototypes may have unanticipated conversions; etc. At the very least, macros, enumeration constants, declarations, etc. may remain unused or declared redundantly. Often such inconsistencies can be the tip-off to a more serious problem.

Some features of C++ lend themselves to a static analysis that was not possible with C. The value of data members can be tracked, at least within member functions. Constructors can be presumed to operate on virgin data allowing us to report on the use of not-yet-initialized data members and the fact that data members have not been initialized at all. Destructors are places where resources should be freed allowing us to report on undeleted pointers.

Finally, C++ itself offers sufficient complexity (to somewhat understate the case) that proper use of its many facilities is not always easily achieved, especially by the novice. Literature exists that alerts users to improper usage **[12**, **13**, **17**, **19**, **20**, **21**, **22]** and many of these recommended do's and don'ts have been incorporated in this package.

However, the proof, as they say, is in the pudding. A simple test that will determine whether this lint or any lint is useful. Simply apply it to your programs and see what turns up.

2.3 Language Definition

The Kernighan & Ritchie (K&R) description of the C programming language **[1]** served as an early de facto standard. An excellent exposition of this standard was provided by Harbison & Steele **[3]**.

During the 1980's, the ANSI (American National Standards Institute) C committee (X3J11) developed a C standard that is largely upward compatible with K&R (one of its major tenets was to "not break working code"). Most major vendors have adopted the standard. ISO (the International Standards Organization) has adopted the ANSI C specification **[2]** known as C90 and authors K&R and H&S have produced subsequent editions of their respective works based on

this ANSI standard. Recently that standard was amended and upgraded to a new standard **[4]**, which is known as C99.

In parallel with the C standardization efforts, Bjarne Stroustrup and others were "pushing the envelope" of enhancements to the C language especially in the areas of abstract typing and "object-oriented" programming. This enhanced version, known as C++, received widespread support in the C community and standardization efforts by ANSI (X3J16) and ISO (WG21) began in 1991. The base documents for the standardization effort were the ANSI C standard **[2]** and the Annotated C++ Reference Manual (ARM) **[11]**. There efforts led to an international standard for C++ **[10]**. Numerous fixes and clarifications were made resulting in a 2003 version **[34]**

For C modules, PC-lint/FlexeLint assumes the C99 definition of C. Via option the user can specify the earlier C90 standard. The product supports K&R where it does not conflict with the selected C standard. It also supports common extensions to the standard especially for specific compilers. See Chapter 14. NON-STANDARD EXTENSIONS, Section 5.8 Compiler Adaptation and Chapter 15. PREPROCESSOR for more information on common extensions.

3. GETTING STARTED WITH PC-LINT

FlexeLint Users: Getting started with and using FlexeLint will depend on your operating system and compiler. Please see the FlexeLint Installation Notes accompanying this document.

3.1 Setup

This section is appropriate for Windows 95, Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP and Vista.

The distribution disc (CD-ROM) should automatically enter a setup program when the disc is placed in the drive. Should this not occur then execute the **pclint9setup.exe** (herein after called "**SETUP**"), directly from the root directory of the CD.

SETUP is a traditional setup/installation program, which verifies that there is sufficient space on the destination drive and then will copy files into the directory you select. We will refer to this directory as the Installation Directory. After installation, it will have the following contents:

lint-nt.exe	PC-lint executable.
config.exe	Configuration program for PC-lint.
readme.txt	Addendum to the manual.
pc-lint.pdf	On line manual in Portable Data Format.
pr.exe	A convenient printing utility.
msg.txt	ASCII rendition of our message descriptions.
unwise.exe	An uninstall program.
install.log	An installation log.
Lnt	A subdirectory containing ...
co-….lnt	Compiler options files for specific compilers.
co.lnt	generic compiler options file.
sl-….c	Standard library modules for K&R compilers.
sl.c	A generic standard library module for K&R compilers.
env-….lnt	Options files for various environments including Microsoft's Visual Studio and various editors.
lib-….lnt	Options files for special 'challenge' libraries.
au-….lnt	Options files for author recommended checks.
Test	A subdirectory that houses various test files. See Section 3.3 Running the Test Programs

At the conclusion of **SETUP**, you will have the option to check a box "I want to start the **config.exe** file now". You should probably answer 'Yes' as configuration is necessary to select an appropriate set of options for your compiler and libraries.

3.2 Configuration

The configuration program (**config.exe**), which we refer to as the configuration wizard (or wizard for short), has two basic modes of operation:

(i) As a wizard to assist you in selecting options appropriate to your compiler, libraries and personal programming preferences. This mode is used after initial installation and any time in the future when conditions change.

(ii) As a program to be run at a later time to allow you to conveniently switch among previously prepared configurations.

In either event, you will be prompted for a directory. We need to distinguish between two directories (which may in fact be the same):

Installation Directory: where PC-lint is installed.

Configuration Directory: where configuration information is stored.

In the case that you have installed PC-lint on your own personal computer such that the installation directory is writable, the configuration and installation directories can be, and indeed probably should be, the same. On the other hand, if you have a LAN license and PC-lint is installed on a read-only file server, the configuration directory will need to be different than the installation directory.

PC-lint does not have a visual front end. It is designed to be run from the command line or from within an integrated development environment or smart editor (See Section 3.5 Integrating With Your Environment). The wizard will create a file called **lin.bat** that you may use to lint C and C++ files using the command:

```
lin options  file1  file2  ...
```

Presumably, **lin.bat** has been placed in your **PATH** either through you or the wizard copying the file into a directory that is already in your **PATH** or by you adding the Configuration Directory to your **PATH**. There is no way for the wizard to add a directory permanently to your path. If requested, it will produce a batch script in a file called **LSET.BAT**, which you can run before linting, to alter your **PATH**.

lin.bat will contain a command similar to the following:

```
InstDir\lint-nt  -iConfDir std.lnt &1 &2 ...
```

where *InstDir* is the Installation Directory, *ConfDir* is the Configuration Directory, and **std.lnt** will contain options appropriate for linting.

The **-i** option (fully described in Section 5.7 Other Options) is so named because it can specify include directories and hence governs file searching. File searching is actually employed for all files and not just headers. In particular, **std.lnt**, which is placed in the Configuration Directory, is designed to be found this way. However, note that a **std.lnt** in your current directory will take priority and this too has its uses.

std.lnt will incorporate a compiler options file, zero or more library options files, size options and a reference to a file called **options.lnt** which is intended to reflect your long-term error suppression strategy. See Section 16.2 Recommended Setup

To summarize, the wizard will create the following files:

LIN.BAT	batch file which can also be copied into your **PATH** and which contains a reference to **STD.LNT**.
STD.LNT	An indirect lint file that will "point to" a compiler options file and an options file and optionally other files and options.
STD_...LNT	sequence of different versions of **STD.LNT** that the wizard can choose from later.
OPTIONS.LNT	A convenient centrally located options file that you will tend to use for most of your linting.
LSET.BAT	A batch file optionally generated to be used to place the PC-lint executable in your **PATH**.

Use of these files is described in Section 3.4 Linting your Programs and Section 16.2 Recommended Setup.

The wizard will also copy a number of files from the installation directory (or its subdirectories) into the configuration directory. The following is typical of the sorts of files you may expect to see:

co-….lnt	Compiler options file(s)
env-….lnt	Environment options files
lib-….lnt	Library options files
lib-….h	Headers referenced by **lib-….lnt** files.
au-….lnt	Options files for author recommended checks
file?.cpp	A pair of test programs described in the next section.

3.3 Running the Test Programs

For PC-lint, you have arrived here either after running **SETUP** and **CONFIG** or after having run **INSTALL**. In the former case, the PC-lint executable will be **lint-nt.exe** and in the latter case, it will be one of **lint.exe** or **lint-os2.exe**. In our examples, we will employ **lin.bat**, which has been artfully configured to contain the appropriate executable.

For FlexeLint you will arrive here after creating an equivalent Shell script (or equivalent command script), which for presentation purposes we assume is named **lin**. In this section, to describe the basic operation of the product we assume two small files **filea.cpp** and **fileb.cpp** defined below. These may be found in directory **dos-ins\test** of the PC-lint distribution disc or you might find it less trouble to type them in fresh.

From your configuration directory type

```
lin filea fileb
```

This applies PC-lint to the sample files **filea.cpp** and **fileb.cpp** shown below.

```
// filea.cpp:
//lint -w2  reduce the warning level to 2

class X
    {
    static int a;
    };


// fileb.cpp:
class X
    {
    int a;
    };
```

You should see on your screen approximately the following message:

```
PC-lint for C/C++ (NT) Vers. 9.00, Copyright ...

--- Module:   filea.cpp (C++)

--- Module:   fileb.cpp (C++)
  _
  };
fileb.cpp  5  Warning 631: tag 'X' defined differently at line 4, file
   filea.cpp
filea.cpp  4  Info 830: Location cited in prior message

--- Global Wrap-up

Warning 1527: static member 'X::a' (line 6, file filea.cpp) not defined
filea.cpp  6  Info 830: Location cited in prior message
```

Not everything you see is an error message. The opening line identifies the version of PC-lint (and the version of the DOS extender if you are running the DOS executable). The 2nd and 3rd lines are in-progress messages, which normally just display the names of the modules being

processed. They are essentially a way of giving a progress report when PC-lint/FlexeLint is processing many files and nothing else is being reported. You can turn off these 'verbosity' messages or increase their detail with the **-v**... option.

The 4th line is the start of an error message. Normally, an error message will begin with a pair of lines. The first line contains only the character '_' which points (downward) to the place on the second line (the actual line of source code) where the error was first encountered. This is followed by a line containing, in order, the name of the file being processed, the line number, the kind of message (Error, Warning, Informational, Elective Note) a message number and a brief description. It is important to note that a more complete description can be found by looking up the message number toward the end of this manual (Chapter 19. MESSAGES).

In many cases (including this example) a prior point in the program is referenced. The name of the file and, possibly, the name of the module are given. (If the error had occurred in a **#include** file, the filename would have been different from its module name and hence both would have been given).

You will note in particular Message **830** which is issued for the sole purpose of allowing an automatic error processor to go to the location cited within the immediately preceding message.

Any option can be placed within a file within a special C or C++ comment as is shown in **filea.cpp**. The only thing special is that the comment begins immediately with the word **lint**. As an experiment, remove the special nature of the comment by introducing a blank before the word **lint**. You have changed the warning level to 3 (the default) and a richer set of diagnostic messages will issue forth.

There are ways of customizing the format of error messages, especially for the automatic processing of messages. See Section 5.6 Message Presentation Options

It is not normally necessary to perform further tests, but running the remaining tests can be instructive.

Create a convenient directory for the purpose and copy in all the files from ***InstDir*\test**. (If you have used **INSTALL** to get here (See Section 3.7 Other Operating Systems), rather than **SETUP**, you should copy the files from the distribution disc in directory **\dos-ins\test**.)

For every file of the form

```
test?.lnt
```

there will be a corresponding output file named:

```
test?.out
```

To run the first of these type at command level:

```
lin  test1.lnt
```

and you should observe on the screen information identical to **test1.out**.

Other test suites are run similarly.

3.4 Linting your Programs

To lint your own programs, enter a directory containing your C and C++ modules. We assume a **lin.bat** (or equivalent shell script in the host operating system) that can be run from any directory (i.e., has been placed in your **PATH**). This has the advantage of simplifying the presentation and perhaps simplifying the typing.

Assuming your program consists of modules

```
alpha.cpp
beta.c
gamma.c
```

we could lint the entire program with

```
lin  alpha.cpp  beta.c  gamma.c
```

Note that an extension of **.cpp** (as well as **.cxx**) normally indicates a C++ module, an extension of **.lnt** indicates an indirection file (command line extension) and all other extents indicate C modules.

Often it is better to first do a unit checkout (using the **-u** option) on separate modules. For example:

```
lin  -u  alpha.cpp
```

does a unit checkout for module **alpha.cpp**. (Any subset of your files may be subjected to unit checkout.) When doing a unit checkout, certain messages are suppressed (such as "function not used").

3.4.1 Other File Extensions for C++ Modules

To override the default treatment of file extensions you may use the **+fcp** flag. This indicates that the next module (and subsequent modules) will be a C++ module. This will remain in effect until turned off with **-fcp**. Thus

```
lin  +fcp  a1.c  a2.c  a3.c  -fcp  a4.c  a5.cpp
```

will treat **a1.c**, **a2.c** and **a3.c** as C++ modules, **a4.c** as a C module and **a5.cpp** as a C++ module.

The option **+cpp(*extension*)** can be used to add to the list of C++ extensions. Thus

```
lin   +cpp(cc)  a1.cc  a2.cc  a3.c  a4.cpp
```

will process **a1.cc**, **a2.cc** and **a4.cpp** as C++ modules and **a3.c** as a C module.

Options are strictly order-dependent.

3.4.2 Controlling the Messages

For previously unlinted code, you may get a discouragingly large number of messages. The default warning level is 3, meaning that you see all Errors, Warnings and Informational messages. To inhibit Informational messages, reduce the warning level to 2 with **-w2** as in

```
lin  -u  -w2  alpha
```

(**.cpp** is assumed). Messages may also be suppressed by message number. For example, to suppress Warning **547** you may use the option **-e547** as in:

```
lin  -u  -e547  gamma.c
```

In any event, do not get discouraged. There are numerous error-suppression options (See Section 5.2 Error Inhibition Options). For advice on establishing an error suppression policy, see Chapter 16. LIVING WITH LINT.

3.4.3 Options

In case you haven't noticed yet, there are numerous options, and a lengthy chapter devoted to them (Chapter 5. OPTIONS). To obtain an option summary type

```
lin  ?
```

3.4.4 Extending the Command Line

For projects of more than one module, you will find it convenient to place the names of all the modules in a lint indirect file. Let us suppose the name of this project is alphabet. Create a file **alphabet.lnt** containing:

```
alpha.cpp
```

```
beta.c
gamma.c
```

then use:

```
lin  alphabet
```

to lint the entire project. You may place options within **alphabet.lnt** but this is not recommended because if you then do a unit checkout on individual modules, you will not receive the benefits of these options. Better, place project-independent options in a centralized file; we have started you off with the file **options.lnt** for this purpose. Project-specific options can be placed in a file called **std.lnt** in your project directory. This **std.lnt** should be an augmentation of the centralized **std.lnt**. This is further described in Section 16.2 Recommended Setup.

3.5 Integrating With Your Environment

PC-lint has been flexibly designed so that you should have no difficulty integrating it with your favorite Integrated Development Environment (IDE), or smart editor. The advantages are that you will be able to launch PC-lint with a minimum number of keystrokes to process your current file or your current project. Upon completion, moreover, you will be able, in most cases, to quickly and automatically sequence from error to error (actually message to message) with hopefully a single keystroke and with your own editor appropriately positioned in the file that needs attention.

We, of course, could have provided our own environment but we felt you probably would prefer to continue using your existing environment with your favorite editor. Hence, we have focused our attention on facilities to bring that about. In particular, our message formatting is extremely flexible (See Section 5.6 Message Presentation Options). Message redirection is available not only via the standard redirection character (**>**) on the command line but may also be embedded as an option (see **-os(*file*)** in Section 5.7 Other Options). The banner line can be controlled (see **-b** in Section 5.7 Other Options) and it can be arranged so that at least one message is always produced (using option **+e900**) which is necessary for some environments. In addition, indirect files (**.lnt** files) can themselves contain indirect files. The file name extension, by which an indirect file is known as such, can be something other than **.lnt** (See **+ext** in Section 5.7 Other Options). Indirect files can contain embedded environment variable names (Section 4.1 Indirect (.lnt) Files).

We have added a number of files of the form **env-...lnt** that assist in the process of integration with particular IDE's. These files are updated as needed. See our web site (under Version 9.00 patches) for the most current.

Instructions for using these **.lnt** files appear as comments within the files themselves. Simply print them out and follow the directions.

3.6 Project Files

Many IDE's (Interactive Development Environments) support the notion of a project file which is an ASCII file that contains module names, defines of pre-processor variables, and include paths. We support the following project file encodings.

***name*.vcproj**	Microsoft Visual Studio 7, 8, and 9
***name*.vcp**	Earlier Microsoft Windows CE
***name*.dsp**	Microsoft Visual Studio 6
***name*.bpr**	Borland

For example, the following creates a lint project file from a **.vcproj** file.

```
lint-nt  project.vcproj >project.lnt
```

To employ such a **project.lnt** file simply use it in the usual fashion. E.g.

```
lin project.lnt
```

The output file (**project.lnt**) will contain useful options as well as a list of modules. For unit checkout, you may want the list of options but not the list of modules. For this purpose, use the **--u** option (as opposed to **-u**). For example:

```
lin --u project.lnt alpha.cpp
```

This will do a unit check using all the options within **project.lnt** but none of the modules. Only **alpha.cpp** will be processed.

Additional information on project files may be found in appropriate **env-...lnt** files.

3.7 DOS and OS/2

We have provided on the distribution disc an INSTALL program that is intended for OS/2, or DOS systems able to support DOS-extended technology.

For operating systems other than these and other than Microsoft Windows, you will be using FlexeLint. For FlexeLint installation and getting started, see the FlexeLint Installation Notes.

To use the INSTALL program, place your distribution disc into a CD-ROM drive, and type:

```
Drive-letter:\DOS-ins\install
```

Follow the instructions on the screen.

The **INSTALL** program does not actually copy files from the distribution disc nor does it modify your environment variables and certainly not your **AUTOEXEC.BAT** or **CONFIG.SYS**. Rather it creates batch files that will do these things for you. In this way, you can see in advance what will happen when you do the copy and you will have a record of what transpired if some question arises later. Also, many programmers have a suite of **AUTOEXEC.BAT** files and need to know how to modify each one of them and not have the current one blindly modified.

The batch files created are:

LCOPY.BAT	Batch file to copy files from the distribution disc to the lint directory.
LSET.BAT	(**LSET.CMD** for OS/2) Batch file to set environment variables (created only if needed). Note: Under DOS, **LCOPY** calls **LSET** upon termination (if needed).

The **INSTALL** program will also create the following files:

LIN.BAT	(**lin.cmd** on OS/2) A batch file, which **INSTALL** will copy into your **PATH** and which contains a reference to **STD.LNT**.
STD.LNT	An indirect lint file that will "point to" a compiler options file and an options file.
STD_...LNT	Possibly multiple versions of **STD.LNT** (See Section 3.7.1 Multiple Configurations)
OPTIONS.LNT	A convenient centrally located options file that you will tend to use for most of your linting.

Use of these files is described in Section 3.4 Linting your Programs and Section 16.2 Recommended Setup

3.7.1 Multiple Configurations

The **INSTALL** program for OS/2 and DOS serves a dual purpose. First, it serves the purpose of copying files from the distribution medium to a hard drive (which is done by **SETUP** for the various Windows platforms). Next, it can create one or more configurations (as embodied in **STD.LNT** files).

Setting up for multiple configurations (i.e., multiple compilers, and/or multiple memory models and/or multiple libraries) is done by establishing multiple candidates for **STD.LNT** and providing an easy way to switch between them. See Section 16.2 Recommended Setup to see the central role played by **STD.LNT**.

If, during your install, you answer "Yes" to the question:

> "Are you going to set up multiple compilers ... "

then the questions about compiler, memory model and libraries will be repeated for as many times as you request. The set of answers that you give to these questions are placed into a sequence of files whose names are:

```
STD_A.LNT
STD_B.LNT
STD_C.LNT
...
```
etc.

As always, **LIN.BAT** contains a reference to **STD.LNT**, which is initially identical to **STD_A.LNT**.

To switch between configurations we provide a batch file called **CONFIG.BAT**. This can be run from your working directory using the full path name (no arguments are needed), which may look like:

```
C:\LINT\CONFIG
```

This will provide a menu of configuration choices. Select one by choosing the appropriate letter. This letter will determine which of the files of the form **STD_*x*.LNT** will replace **STD.LNT**.

You, of course, do not have to use **CONFIG** in this way. For example, if, in a particular directory, you always use the 'C' configuration (i.e., **STD_C.LNT**), you may create a file in that directory called **STD.LNT**, which contains a reference to the file:

```
STD_C.LNT
```

Alternatively, you could simply copy the contents of file **STD_C.LNT** from the lint directory to the file named **STD.LNT** in the given directory.

3.7.2 DOS-ins Files

Running **LCOPY** will copy files from the directory **DOS-ins** on the distribution disc. The most prominent file is the Lint Executable. Typically only one executable of the following three is copied.

lint.exe	DOS version with a built-in DOS extender
lint-nt.exe	Windows version
lint-os2.exe	OS/2 Version

lint.exe has been bound with the Phar Lap 386/DOS-Extender. You do NOT require a separate license to run Phar Lap nor separate software. It does, however, require an 80386 (or better) machine.

lint-nt.exe is a 32-bit console application that runs on Microsoft Windows. This is identical to the executable installed via SETUP and if this is the executable you obtained you may have made a mistake since the recommended installation for these operating systems is SETUP.

lint-os2.exe is a 32-bit OS/2 version.

Other files in the **DOS-ins** directory that may be copied are:

readme.txt	Supplementary information to this document.
pr.exe	A convenient printing utility.
choose.exe	A utility used by CONFIG.BAT described in Section 3.7.1 Multiple Configurations.
msg.txt	An ASCII file describing the lint messages (essentially Chapter 19. MESSAGES).

In addition to files in **DOS-ins** there are three subdirectories from which files may be copied:

```
DOS-ins\lnt
DOS-ins\test
DOS-ins\pharlap
```

Subdirectory **DOS-ins\lnt** contains the following files:

co-...lnt	Compiler options files for specific compilers.
co.lnt	A generic compiler options file.
sl-...c	Standard library modules for K&R compilers.
sl.c	A generic standard library module for K&R compilers
env-...lnt	Options files for various environments including the Borland/Turbo IDE and Microsoft's Visual Studio.
lib-...lnt	Options files for special 'challenge' libraries.
au-...lnt	Options files for author recommended checks

Subdirectory **DOS-ins\test** contains testing files, some of which are copied during installation.

```
filea.cpp  test1.lnt  module1.cpp
fileb.cpp  test2.lnt  module2.c
test.h     test1.out  module3.cpp
           test2.out  module4.cpp
```

See Section 3.3 Running the Test Programs below on how to use them.

Subdirectory **DOS-ins\pharlap** contains files used to support the Pharlap DOS extender. See Section 3.7.3 DOS Extender Notes.

3.7.3 DOS Extender Notes

When PC-lint was first developed, the operating environment was MS-DOS with its dreaded 640K DOS limit. Relief from this limit came in a variety of ways: OS/2, DOS extenders, and more recently, Windows. For much of the early time period during which PC-lint has been commercially available, the primary method of obtaining large address space was through the use of the Phar Lap 32-bit DOS extender.

LINT.EXE is bound with the Phar Lap 386 DOS extender. This will run under MS-DOS, Microsoft Windows (though **LINT-NT.EXE** is recommended for Windows 95 and later) and OS/2 (Version 2.x). It will run under the memory managers QEMM, 386MAX and HIMEM.SYS. It supports all the key industry standards for memory management including EMS, XMS, VCPI and DPMI. The software is remarkably robust but nonetheless field problems can occur.

If you suspect a difficulty with the DOS extender you should copy the contents of the **DOS-ins\pharlap** subdirectory onto your hard drive. This contains the following files:

tellme.exe	A utility provided by Phar Lap to describe the system that you are using in case things go wrong.
cfig386.exe	A configuration utility provided by Phar Lap to configure the DOS extender embedded within PC-lint and to be used only in case of field difficulties.
***.txt**	Files whose names end in **.txt** have been provided by Phar Lap to describe these utilities.

Some users have reported difficulties running the DOS-extended PC-lint (**LINT.EXE**) with DOS 6.00 and **EMM386.EXE**. With a **CONFIG.SYS** as simple as:

```
device=c:\dos\himem.sys
device=c:\dos\emm386.exe
```

they may experience long startup times (5-7 seconds before the banner line would be printed) or sometimes the machine would hang and/or reboot. Moreover, these difficulties do not appear when running from within Windows.

If you experience any of these symptoms you may find running the following program helpful:

```
C:\LINT>cfig386 lint.exe -maxv 2000000
```

this will have the effect of limiting the amount of memory available to PC-lint to approximately 2 Megabytes. Adjust the numerical quantity to suit the memory requirements of your linted application (The **-vs** option may be helpful here).

LINT.EXE is initially set up to operate with normal DOS expectations, i.e., **int** and near pointers are 2 bytes with far pointers 4 bytes. **LINT-NT.EXE** and **LINT-OS2.EXE**, by contrast, assume a 32-bit flat model where **int** and pointers are both 4 bytes. To lint 386 native code programs (with any version of PC-lint) use the options:

```
-si4     -spN4     -spF6
```

To lint DOS programs with any version of PC-lint use the options:

```
-si2 -spN2 -spF4
```

See Section 5.3 Size and Alignment Options

4. THE COMMAND LINE

The command line for PC-lint/FlexeLint has the form:

```
lint  options  file1  [ file2  file3 ... ]
```

where the command name **lint** will vary depending on the operating system and installation choices; ***options*** (described in Chapter 5. OPTIONS) generally precede the files as shown here. They may also be interspersed among the files to achieve special effects, as options and files are processed together in left to right order. If no arguments are given to PC-lint/FlexeLint or if a **?** is provided as a single argument, a short summary of help information is displayed on standard out.

A filename is any name that can be passed to **fopen**. Also, "wild card" characters are permitted in PC-lint and in FlexeLint for Unix. Thus, ***.c** represents all **.c** files in a given directory.

Files are either C or C++ source files (modules) or indirect files (**.lnt** files). Indirect files may contain more options and files. See Section 4.1 Indirect (.lnt) Files

A module is considered a C++ module (as opposed to a C module) if its name has an extension of **.cpp** or **.cxx** or if the **+fcp** flag is ON. You may add to the list of C++ designation extensions using the **+cpp** option (See Section 5.7 Other Options).

Please note that you would not normally place the names of header files on the command line. Header files are processed when they are encountered by **#include** lines within other source files.

If a file extension is omitted, the extension **.lnt** is tried first. If the file is not found, the extension **.cpp** is tried, and finally the extension **.c** is tried. See **+ext(...)** option in Section 5.7 Other Options.

Files are subject to the same search algorithm as quote-style **#include** files (See Section 15.2 include Processing), which is especially handy for indirect files.

If the environment variable **LINT** is set, the associated value is prepended to the arguments on the **lint** command line with an assumed intervening blank. For example:

```
set LINT=-voif
```

has the effect of setting the **LINT** environment variable to "**-voif**". As we will see later, this is a verbosity option that displays all options, and names of all files processed including indirect files.

PC-lint/FlexeLint will return an exit code (See Section 4.2 Exit Code).

4.1 Indirect (.lnt) Files

If the extension is **.lnt** or equivalent (see option **+lnt** in Section 5.7 Other Options), the file is taken as *an **indirection***, in which case it may contain one or several lines of information that would otherwise be placed on the command line. Indirect files may contain indirect files to any depth. Indirect files and other files may be interspersed in any manner desired. Indirect files may contain comments in addition to options and files. Both the standard C-style comment, **/*** ... ***/** and the C++ style of comment, **//** ... ***end-of-line***, are supported.

An indirect file may also contain environment variables (where supported by the operating systems) delimited by '**%**' characters for most operating systems. Thus an indirect file containing

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

employs the environment variable **SOURCE** to specify where files are coming from. The environment variable specification is case sensitive.

If an indirect file is not found in the current directory a search is made in the usual places. As an example **lin.bat** may contain

```
C:\lintpp\lint  -iC:\lintpp  std.lnt  %1 %2 %3 %4
```

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

std.lnt could contain:

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

This illustrates the nesting of indirect files.

4.2 Exit Code

The operating system supports the notion of an exit code whereby a program may report a byte of information back to a controlling program. PC-lint/FlexeLint will set the exit code to the number of messages reported (with an upper bound of 255). Thus, suppressing messages will lower the exit code. If you want the exit code to always be **0** use the option **-zero**. If you don't want all of the messages to be counted, use **-zero(#)** as discussed in Section 5.7 Other Options.

For most operating systems, you may use the exit code within batch procedures to conditionally alter the flow of control. For example:

```
lint ... usual arguments ...
if errorlevel 1 goto errorsfound
```

within a batch procedure will cause a jump to label **errorsfound** if one or more errors are detected by PC-lint/FlexeLint. This could be used to jump around a compilation.

Non-zero exit codes are commonly used by **make** programs to terminate processing.

5. OPTIONS

5.1 Rules for Specifying Options

Options begin with a plus (**+**) or minus (**-**) sign except for **!** as noted below. They may be interspersed among the filenames throughout the command line (and within an indirect file). They are processed in order meaning that an option specified after a filename will not take effect until after that file is processed.

Options within Comments

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

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

or

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

Options within comments should be blank-separated (but may under special circumstances contain blanks -- See Section 5.1 Rules for Specifying Options). The optional commentary should, of course, not begin with a plus or a minus or an exclamation point. Note that the '**lint**' within the comment must be lower-case as shown and must lie immediately adjacent to the '**/***' or '**//**'. The **/*lint** comment may span multiple lines. Note that within indirect files (**.lnt** files), options appear just as on the command line. They need not and should not be placed within a **/*lint** comment.

PC-lint/FlexeLint also supports the Unix in-code options such as

```
/* LINTLIBRARY */
```

See Section 13.9 Unix Lint Options

Options within Macros

It is also possible to place options within a macro. If a macro contains a lint comment (one of the form **/*lint** ***option***(s)...) then the comment and option(s) take effect when the macro is expanded. To take a somewhat extreme example, the macro below allows for division by zero without complaint from lint.

```
#define DIVZERO(x) /*lint -save -e54 */ ((x) /0)  /*lint -restore */
```

Indeed, this would be done automatically for you, were you to use the option **-emacro(54,DIVZERO)**.

At this writing there is an 80 character limit on the length of a lint comment embedded within a macro. Thus a comment such as

```
/*lint -save -e54 */
```

may not exceed 80 characters. Exceeding this limit will result in a fatal diagnostic (message **323**). If you have a long sequence of options, simply decompose the sequence into two or more **/*lint** options.

Macros may not contain one line error suppression options, i.e. of the form **!e**....

Blanks within Options

Since options are blank separated, blanks may not appear within options unless they appear within parentheses or are quoted. Even then, a blank-bearing option placed on the command line may be inadvertently split by the operating system shell (command-line parser) whose designers may not have read this manual. Hence, our discussion about blanks in options applies only to options appearing in ***indirect files or in lint comments***, which are beyond the control of the system argument crackers.

A number of our options are specified in terms of parenthetical arguments. With these, interior blanks adjacent to the three characters **),(** are ignored. By interior we mean within parentheses. Thus:

```
-esym(534,printf,scanf,operator new)

-esym(534, printf, scanf, operator new)

-esym( 534 , printf , scanf , operator new )
```

are all equivalent, but are different from

```
-esym(534,printf,scanf,operator  new)
```

The last has two blanks separating **operator** from **new**. This incidentally is certainly an error because the built in function **operator new()** has one blank separating the two keywords. The same ignore-blank considerations apply recursively to sub-lists as used, for example, in the **-function** option. Thus the following two options are equivalent:

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

You may also employ double quote (") -- but not single quote (') -- to protect a blank. For example, the option

```
-"dWORD=unsigned short"
```

is like placing

```
#define WORD unsigned short
```

at the head of each module. You may place the initial quote anywhere after the leading '-' and before the '=' to achieve the same effect. If you place the quote before the '-', the construct will not, in general, be taken as an option. (If the operating system strips off the quotes for you, it may yet work.) If you place the quote after the '=' sign or after a left parenthesis, the quotes will NOT be dropped. This has a purpose. Thus

```
-dMESSAGE="use unsigned short"
```

is like placing the option:

```
#define MESSAGE "use unsigned short"
```

at the head of each module.

Option Display

Because options can be placed in obscure places and then forgotten, the verbosity option **-vo**, (See Section 5.4 Verbosity Options) can be used to display all options as they are encountered.

Options and Meta Characters

Some of our more complex options have the form

***-identifier*(*arg1*, *arg2*, ...)**

These options can always be placed in **.lnt** indirect files and in **/*lint** and **//lint** comments. On the command line, however, such options can bear characters that conflict with a command interpreter's meta characters. Depending on the operating system and/or command interpreter, these characters can include the comma, the parentheses, and question mark.

One approach is to employ command interpreter conventions to escape the meta characters. For example, most command interpreters will allow you to quote an argument (and will strip off the quotes before passing them to the command). An example of this is:

```
lint "-esym(714,alpha)"  x.cpp
```

This technique will even allow you to pass blanks into PC-lint/FlexeLint. E.g.

```
lint "\program files\a.cpp"
```

If the quote method is unavailable or undesirable, you may use alternatives to the more troublesome characters. On MS-DOS, the comma character is not suitable in an option when a batch command is employed, as it is taken as a separator. The exclamation point (**!**) can be taken as an alternative to the comma as in:

```
lin -esym(714!alpha) a.cpp
```

On the various flavors of Unix, parentheses and question marks are taken as meta characters. PC-lint recognizes square brackets and the period respectively as alternatives to parentheses and the question mark. E.g.

```
lint -e75.  -esym[714,alpha]
```

is taken to be the equivalent of

```
lint -e75? -esym(714,alpha)
```

It should be stressed that these characters are only supplementary alternatives. It is better to use the originals within files to promote file communication among systems.

5.2 Error Inhibition Options

Options beginning with **-e** enable the user to inhibit error messages. The same option beginning with **+e** restores the message. All messages except the Elective Notes, (**900** level and **1900** level) are ON by default. Inhibiting a message (or message class) does ***not*** affect lint processing other than to suppress the outputting of the message.

Additionally, ***-wlevel*** (in Section 5.7 Other Options) can alter the warning 'level' and, hence, have the effect of inhibiting messages. Also, see Chapter 16. LIVING WITH LINT for hints and guidance in forming an error suppression policy.

Note that there is no option to suppress a message within some file. This is typically something that programmers want to do when they have no control over some header file. To do this make sure the header is classified as a ***Library*** header. The default process of categorizing headers as Library works well most of the time but there are a variety of options to fine-tune the process (See Chapter 6. LIBRARIES). Once dubbed a library header, there are a variety of options to suppress messages associated with Library headers (**-elib**, **-elibsym**) and an option, which independently sets the warning level within Library Headers (**-wlib**). Most of the compiler options files supplied with our product sets this warning level to 1.

-e# (where # is a number or numeric pattern) inhibits, and
+e# re-enables, error message(s) #.
For example **-e504** will turn off error message **504**. The number designator may contain 'wild card' characters '**?**' (single character match) or '*****' (multiple character match). For example **-e7??** will turn off all **700** level errors. Note: to turn off Informational messages it is better to use **-w2** (See Section 5.7 Other Options).

As another example:

```
-e1*
```

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

-e(#[,#]...) will inhibit message number(s) # for the next expression.
(Presumably this is used within a lint comment). For example:

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

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

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

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

Not only is the former method simpler but it is more effective. There are two phases to expression analysis: a syntax analysis, which produces an expression tree and a semantic analysis, which walks the tree. Expression trees can be saved and walked later. This is done for the controlling clauses of the **for** and **while** statements. In the case of the **-e(#)** option, the error suppression is placed within the expression tree so that it is available for later evaluation. With the **-save**, **-restore** method, it is not.

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

--e(#[,#]...) will inhibit message number(s) # for the entire expression before or within which it is placed. For example:

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

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

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

-e{ *#* [**,** *#*] …**}** will inhibit message number(s) *#* for the next statement or declaration (which ever comes first). This is presumably used within a lint comment. Consider the following example:

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

void f( int n, unsigned u, int k )
    {

    //lint -e{732}  suppress "loss of sign"
    u = n;        // 732 not issued

    //lint -e{713}  suppress "loss of precision"
    if( n )
       {
       n = u;    // 713 not issued
       }
    }             // 715 not issued
```

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

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

The use of **–e{***#***}** is to be preferred over the seeming equivalent:

```
-save –e# … -restore
```

This is not only because it is simpler but also because inhibition of the form **–e{}** is embedded in any statement tree that is constructed and so the inhibition remains in effect during the Specific Walk as well as the General Walk.

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

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

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

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

then,

```
Suppress(715)
```

will suppress message **715** for the next statement or declaration.

--e{ *#* [**,** *#*] … **}** will inhibit message number(s) *#* for the entire braced region in which it is placed. A braced region may be a compound statement, a function body, a class, struct, or union definition, a namespace body and linkage specification body. If the option is not placed within any such braced region, the suppression applies to the module as a whole (from the point of appearance until the end of the module). It does not extend past the end of the module into the next module. It will affect wrap-up messages for the module but will have no effect on the global wrap-up.

Consider the following example:

```
//lint --e{528}  suppress "f() not referenced"

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

     if( n )
         {
         //lint --e{713}  suppress "loss of precision"
         n = u;    // 713 not issued
         u = n;    // 732 not issued
         }
     }             // 715 not issued
```

The **--e{528}** suppresses the Warning **528** issued at module wrap-up, without affecting other modules. The **--e{715, 732}** suppresses the **715** and **732** issued within the function without affecting other functions. The **--e{713}** suppresses the loss of precision message within its compound statement.

Many of the remarks made about the **-e{}** also apply to **--e{}**. Like **-e{}**, any **--e{}** option is embedded in the statement tree; it can result from a **--emacro({#}** …**)** construct, it may appear in your own macros; and it may use wild cards.

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

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

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

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

For C90 code, the **/*lint** form of comment would be more appropriate:

```
if( x = f(34) )    /*lint !e720 Suppress 'Boolean test of assignment'*/
   y = y / x;
```

Note that C99 does allow '**//**' and PC-lint by default supports C99 conventions. Multiple error message suppression options are permitted as in the following, but not wild card characters.

```
n = u / -1;      //lint !e573 !e721
```

A limitation is that the one-line message suppression may not be placed within macros. This is done for speed. A rapid scan is made of each non-preprocessor input line to look for the character '**!**'. If this option could be embedded in a macro, such a rapid search could not be done.

-ea*letter* Argument Mismatch Inhibition. Note: This affects only non-prototype calls thus it has no effect for C++ programs and decreasing significance for C programs. ***letter*** is one of

- **i** sub-integer
- **n** nominal
- **u** unsigned vs. signed
- **s** same size

This option suppresses warning **516** (argument type mismatch) for selected type differences. Warning **516** is issued when actual arguments and/or formal parameters are inconsistent in function calls not made in the presence of a prototype.

-eai refers to Argument type mismatches of the form **char** vs. **int** or **short** vs **int**. Such a difference can occur when an old-style function definition of **char** (promoted to **int**) meets a prototype of **char**. See also **+fxc** and **+fxs**. This option is recommended only if your compiler always passes at least a full **int** as argument.

-ean refers to Argument type mismatches where the arguments differ Nominally. Examples include the case where one argument is **int** and the other is **long** and where both **int** and **long** are the same size. This also affects argument mismatches between **unsigned int** and **unsigned long** where both are the same size. It can also suppress messages involving **short** and **int** when these are the same size.

-eau refers to type differences where one type is a signed and the other an Unsigned quantity of the same type. For example, if the function **f** expects an unsigned integer and **n** is an **int**, then the call, **f(n)**, will normally draw warning message # **516**. This message will be inhibited if **-eau** is set.

-eas refers to unlike types where both types occupy the same Size. For example, if the function **f()** expects a pointer argument then the call, **f(3)**, will normally draw a message (# **516**). If pointers occupy the same space as integers (they do by default) then the message will be inhibited if **-eas** is set.

-ean is orthogonal to **-eau**; neither option implies the other. If both options are set, **int** match up with **unsigned long**, etc. provided they are the same size. Note that **-eas** implies **-ean** and **-eau**. E.g., if **-eas** is set, it is not necessary to also set **-eau**.

-ecall(*#*,*Name1*[, *Name2* ...]) inhibits
+ecall(*#*,*Name1*[, *Name2* ...]) 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. This includes the parsing of the function call and any of the arguments to the call. Example:

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

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

-efile(*#*, *file* [, *file*] ...) inhibits and
+efile(*#*, *file* [, *file*] ...) re-enables
error message *#* for the indicated files. This works exactly like **-esym** but only on those messages parameterized by ***FileName*** (Eg., **7**, **305**, **306**, **307**, **314**, **404**, **405**, **406**, **537**, **766**). Please note, this does not inhibit messages ***within*** a ***file*** but rather messages ***about*** a ***file***.

There are times when you might want to quote the 2nd argument to **efile** and/or escape some of the pattern valued characters. See **Section 5.2.1 Meta Characters -esym** ...for an explanation and examples.

The current file can be designated using a period ('.') as the second argument. For example:

```
//lint -efile(766,.) This file is being used.
```

designates that the file bearing the lint directive should not have message 766 reported about it.

As another example suppose that any file bearing the preprocessor command **#ident** should be considered as being used. Then the options

```
-ppw_asgn( ident, macro )
-d"sharp_ident()=/*lint -efile(766,.) */"
```

will accomplish the goal without the necessity of modifying any header file. See **#macro**.

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

It is sometimes useful to inhibit error suppression options so that the programmer can view what messages had been suppressed. The option **+efreeze** is designed to do precisely this. After the **+efreeze** option is given we are said to be in a frozen state. In a frozen state the following options will have no effect.

```
-e
!e
-ecall
-efile
-efunc
-elib
-elibcall
-elibmacro
-elibsym
-emacro
-estring
-esym
-etemplate
-etype
-w
-wlib
```

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

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

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

```
lin +efreeze x.c
```

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

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

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

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

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

-/+efreeze(w# [**,**w# ...]**)** Behaves like **-/+efreeze**, but acts only on messages in the given warning level(s). These warning levels are defined by the **-w** option and have the following significance

w0	internal errors and fatal messages
w1	Syntactic Errors
w2	Warnings
w3	Informational
w4	Elective Notes

All of the message suppression options listed above for **+/-efreeze** will be "filtered" such that only message numbers that are not in a frozen warning level may be suppressed or enabled. E.g.:

```
+efreeze(w0,w1) -esym(4*,f)
```

has the effect of suppressing messages 400 through 499 for symbols named "**f**". Messages **4**, and **40** through **49** will remain enabled because messages at warning level one are frozen.

-efreeze(w#**)** reverses the effect of **+efreeze(**w#**)**

++efreeze(w#) locks in freezing permanently for the given warning level

Note: The **+efreeze** option will affect the error suppression of **-emacro()** options even though the **-emacro()** option was given before the **+efreeze** option. The reason for this is that **-emacro()**'s implementation involves the insertion of suppression options into a lint comment that surrounds the macro expansion.

-efunc(*#*, ***Symbol*** [, ***Symbol***] ... **)** inhibits and
+efunc(*#*, ***Symbol*** [, ***Symbol***] ... **)** re-enables
message *#* issued from ***within*** any of the named functions. For example:

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

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

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

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

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

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

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

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

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

There are times when you might want to quote the 2nd argument to **efunc** and/or escape some of the pattern valued characters. See Section 5.2.1 Meta Characters -esym ...for an explanation and examples.

-elib(*#* [, *#*] ... **)** inhibits and
+elib(*#* [, *#*] ... **)** re-enables
error message *#* within library headers or in regions of code denoted as being library. See Chapter 6. LIBRARIES. This is handy because library headers are usually "beyond the

control" of the individual programmer. For example, if the **stdio.h** you are using has the construct

```
#endif comment
```

instead of

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

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

-elibcall(*#***[,***#***])** inhibits
+elibcall(*#***[,***#***])** re-enables
the numbered messages while parsing calls to library functions. This does what **-ecall** does except that it applies to functions that are deemed to be library.

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

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

-elibsym(*#* [**,** *#*] ... **)** inhibits
+elibsym(*#* [**,** *#*] ... **)** re-enables
message(s) numbered *#* that are parameterized by library symbols. For example, suppose a library defines a pair of classes:

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

This will result in message **1510**, base class **X** has no destructor. Note that the message is deferred until derived class **Y** bearing a destructor is seen. The option **-elib(1510)** will suppress this message while processing library headers. If in the user's own code there is a declaration:

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

the diagnostic will be issued in spite of the fact that **-elib(1510)** is given because we are outside library code. The user may suppress this by using

```
-esym( 1510, X )
```

But if there are a large number of such base classes, all lacking a destructor, the user may prefer to issue the option:

```
-elibsym( 1510 )
```

which in effect does an **-esym(1510,*s*)** for all library header symbols ***s***.

-emacro(*#*, ***symbol***, ... **)** inhibits
+emacro(*#*, ***symbol***, ... **)** re-enables
message *#* for each of the macro symbols given. The option must precede the macro definition. The option **-emacro(*#*,*symbol*,...)** is designed to suppress message number *#* for each of the listed macros. For example,

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

will suppress message **778** when expanding macro **TROUBLE**. What actually happens is that when the macro is **#define**'d its defining string is preceded by the lint comment:

```
/*lint -save -e778 */
```

and is followed by the lint comment:

```
/*lint -restore */
```

Please note that

```
 -e123  +emacro(123,A)
```

does not do what you might think. A **+emacro** only undoes a **-emacro** having the same arguments. There is no way currently of enabling a message for only a selected set of macros.

You may wonder why we bother with this option because clearly the user can do this for him (or her) self. But often it is compiler or third-party library macros that need the error suppression and these, many of our users feel, should not be disturbed by the end-user.

There are times when you might want to quote the 2nd argument to **emacro** and/or escape some of the pattern valued characters. See Section 5.2.1 Meta Characters -esym ... for an explanation and examples.

-emacro((#), *symbol*, ...) inhibits, for a macro expression,
--emacro((#), *symbol*, ...) inhibits, for the entire expression,
message # for each of the macro symbols given. The macros are expected to be expressions (syntactically).

There is an occasional problem with inhibiting messages emanating from expressions using the earlier form (in which parentheses do not surround the error number). For example, assume that macro **ZERO** is defined as follows:

```
#define ZERO (*(int *)0)
```

When **ZERO** is actually used, this results in Warning **413** (use of NULL pointer). To suppress this warning, you might think to use:

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

This will not always work owing to the fact that message **413** is given in an expression-tree walk separate from the syntax analysis phase. The fix is to use the following notation:

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

This inhibits **413** during syntax analysis and places a Warning **413** inhibition request in the tree node associated with the macro. What happens is that the option **-e(413)** is placed within a **/*lint** comment before the expression. It is as if we had defined **ZERO** as:

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

You may confirm this experimentally through use of the **-p** option. Consult earlier in this section for a description of the **-e(#)** option.

For full generality you may place a list of error numbers including wild cards between parens. Thus

```
-emacro( (413, 613), ZERO )
```

suppresses the two indicated messages for macro **ZERO**. Also

```
-emacro( (*), ZERO )
```

suppresses all messages for the macro.

offsetof is treated as a special case. For this macro, message numbers are automatically parenthesized. If the user should select as an option, say,

-emacro(413,offsetof) then this will automatically be treated as if the option were **-emacro((413),offsetof)**. The same is true for any other message number.

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

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

would, continuing the earlier example, be as if we had defined **ZERO** as:

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

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

-emacro({#}, *symbol*, …) inhibits for a macro statement,
--emacro({#}, *symbol*, …) inhibits for a macro within a region,
message # for each of the macros given. For example, the macro **Swap** below will swap the values of two integers.

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

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

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

Use of the **--emacro({#}** …**)** option will cause the lint comment

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

to be prepended to the macro.

The example above is not particularly motivating as the same effect could be achieved without the braces. The compelling reason to use the braced form is to control messages during Specific Walks. When the braced form of the **–emacro** option is used, the error inhibition is attached to the statement tree that is produced. When this statement tree is walked during the Specific Walk the curly brace error inhibition will have an effect, as will the **–emacro((#),...)** form, but the plain form will not.

-ep*letter*(*s*) Pointer to Type Mismatch
This option refers to pointer-to-pointer mismatch (Error **64**) across assignment or implied assignment as in initializers, **return** from function, or passing arguments in the presence

of a prototype. By selecting one or more of these options, the user can suppress notification of this error for selected pointer differences. ***letter*** is one of

n nominal
u unsigned vs. signed
nc nominal **char**
uc unsigned vs. signed **char**

s same size
p all indirect values

-epn The pointed-to types differ Nominally. For example, pointer to **short** vs. pointer to **int** where **int** and **short** are the same size.

-epu The pointed-to types differ in that one is an **unsigned** version of the other. For example, pointer to **char** being assigned to pointer to **unsigned char**.

-epnc The pointed-to types differ in that one is a **char** and the other is a **signed char** where **char** are by default **signed**, or where one is a **char** and the other is an **unsigned char** where **char** is by default **unsigned**. This presumes that a distinction is being made between these two types in the first place (**+fdc** is on).

-epuc The pointed-to types differ in that one is a **signed char** and the other an **unsigned char**. This is useful for passing pointers to **unsigned char** to a library routine that may ask for pointers to **signed char** without using a cast.

-eps The pointed-to types differ but they are the same Size. For example, pointer to **long** vs. pointer to a **union** containing a **long** but nothing larger.

-epp The pointed-to types differ in any way imaginable. Said another way... "Pointers are pointers".

-epn is orthogonal to **-epu**; neither implies the other and both can meaningfully be selected. **-eps** implies both. If you select **-eps** you needn't bother selecting **-epu** or **-epn**. **-epp** implies the others.

-epnc is a special case of **-epn** and **-epuc** is a special case of **-epu**. If you specify the latter cases you needn't specify the former.

-estring(*#*,*String1*[, *String2*, ...]) inhibits
+estring(*#*,*String1*[, *String2*, ...]) re-enables
message *#* for the indicated ***Strings*** used as Lint message parameters. Consider the following example:

```
int f()
    {
```

```
        return 20000u << 18;
        }
```

This will result in the following message.

```
                    _
           return 20000u << 18;
x.t  4  Warning 648: Overflow in computing constant for operation:
     'unsigned shift left'
```

If we examine Warning **648**, we see that it is parameterized by a ***String*** in quotes. We may use the **-estring** to suppress on the basis of this string using the option:

```
-estring(648,"unsigned shift left")
```

The second argument to **-estring()** means that the **648** will not be issued when the operation (represented by the '***String***' parameter) is "**unsigned shift left**". Note that we can also disable this message for all bit shifts with:

```
-estring(648,*shift*)
```

The **-estring** option may be used with messages that are parameterized by ***Context***, ***Kind***, ***String***, ***Type*** and ***TypeDiff***. See Message Parameters in **Chapter 19. Messages**.

Note that **estring** participates with other options (like **esym**) in"voting" for or against each message issuance. For example, suppose you're interested in enforcing only one advisory rule from MISRA 2004 which states that expressions should avoid dependence on operator precedence rules, and that you want this to apply for uses of C++ operator functions as well. This implies that we must use

```
+esym( 961, 47 )  // Enforce MISRA rule 47
```

Unfortunately, this means we'll get the following diagnostic for "Hello, world":

```
                                        _
  std::cout << "Hello, world! << std::endl;
  Note 961:  Violates MISRA Advisory Rule 47, dependence placed
             on C's operator precedence; operators:  '<<' and '<<'
```

Suppose you want to suppress this message when **<<** operators are used in tandem as in the example above. This can be achieved with

```
-estring(961, "dependence*on*precedence; operators: '<<' and '<<'")
```

All other violations of rule 47 will still be reported.

-esym(***#, Symbol*** [**,** ***Symbol***] ... **)** inhibits and
+esym(***#, Symbol*** [**,** ***Symbol***] ... **)** re-enables
error message *#* for the indicated symbols. This is one of the more useful options because it inhibits messages with laser-like precision. For example **-esym(714,alpha,beta)** will inhibit error message **714** for symbols **alpha** and **beta**. Messages that are parameterized by the identifier ***Symbol*** or ***Name*** can be so suppressed. Also, if the **fsn** flag is ON (See Section 5.5) messages parameterized by ***String*** may also be suppressed. Thus, if you examine Message **714** in Section 19.5 C Informational Messages you will notice that the italicized word '***Symbol***' appears in the text of the message. See also the entries for '***Symbol***' and '***Name***' in the beginning of Chapter 19. MESSAGES.

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

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

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

```
1:
2:  class X
3:      {
4:      void f(double, int);
5:      };
6:
```

the resulting messages would include:

```
Info 754: local structure member X::f(double, int) (line 4, file
     t.cpp) not referenced
```

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

```
-esym( 754, X::f )
```

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

```
-esym( 754, f )
```

to suppress errors about **X::f**.

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

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

will cause **714** to be reported only for **alpha**. Note that this enhancement may also be used to enable specific MISRA rules specified by Messges **960**, **961**, **1960** and **1963** without adopting the entire set. Simply use the MISRA rule number as a symbol name

```
+esym( 960, 88, 87 )
```

Enables checking for MISRA rules 88 and 87.

Note as a result of the default warning level (3), message **960** is not normally enabled.

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

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

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

5.2.1 Meta Characters for -esym, -efile, -emacro, -efunc, -estring, -etype, -ecall

The following meta characters may be used within names that are used as arguments for **-esym**, **-efile**, **-emacro**, **-efunc**, **-estring**, **-etype** and **-ecall**.

*****	wild card character matching 0 or more characters
?	wild card character matching any single character
‘	backtick used to escape any meta character
[...]	bracketed string meaning optional matches
"..."	used when incorporating comma (**,**) or unbalanced '**(**' or '**)**' with an argument

Although an option containing meta characters may not always be placed on a command line because the special characters may trip up the shell (command interpreter), it can be placed in a **.lnt** file.

Arguments (both error numbers and symbols) may contain 'Wild-card' characters.

For example

```
-esym( 715, un_* )
```

suppresses message **715** for any symbol whose first three characters are "**un_**". As another example:

```
-esym( 512, ?, *::* )
```

suppresses message **512** for any symbol name containing exactly one character and for any name containing a "**::**", i.e. for any member name. As another example:

```
-esym( *, name )
```

suppresses any message about symbol **name**

A string of the form **[...]** means that the string bounded by square brackets is optional. E.g.

```
-esym( 768, [A::] alpha )
```

will suppress message **768** for symbols "**alpha**" and "**A::alpha**". Wild-card characters may appear within the brackets. Thus

```
-esym( 768, [*::]alpha )
```

will suppress **768** for any symbol where name is "**alpha**" no matter how deeply nested within classes and or namespaces it may be. The accent grave character **‘** is sometimes referred to as the backtick. It can be used to escape any of the meta characters. For example

```
-esym( 1533, operator* )
```

suppresses Warning **1533** for any function whose name begins with "**operator**". However

```
-esym( 1533, operator‘* )
```

suppresses **1533** for the function named **operator***.

Within options, commas separate arguments. But suppose your argument contains a comma. For this you may use the double-quote. Example,

```
template <class T, class U> class B;
```

```
//lint -esym(753,"B<<1>,<2>>")
//  Without the double quotes, -esym() sees three arguments:
//  "753", "B<<1>", and "<2>>"
```

Both wildcard characters and non-wildcard characters may be used both within and outside of the double-quoted sequence. A right parenthesis or comma that appears outside of a double-quoted sequence marks the end of the argument as usual. You may use any number of quoted sequences in a single argument. The quote characters will later be stripped away when the option is processed.

The character literals **"**, **‘**, *****, **?**, **[**, and **]** may be expressed by escaping them with a backtick:

```
‘"    ‘‘    ‘*    ‘?    ‘[    ‘]
```

As a special case, the string "**[]**" (as in "**operator[]**" or "**extern int a[];**") need not be escaped since, as a wildcard pattern, it is meaningless.

All escape sequences other than those mentioned above are reserved for future use. Currently, when we encounter such a sequence, we will ignore the backtick and issue a warning. For example, "**‘a**" is taken as "**a**".

[Note: The backslash character is commonly used as an escape character in programming languages such as C and C++ and so it might have seemed more natural to use that instead of the backtick. However, for many users, ’\’ also serves as a directory separator. Since arguments to **-esym()** may now contain path names, and because the backtick is very rarely found in C/C++ source code, this new interpretation shouldn’t result in any additional "version shock".]

There are times when you might want to quote the 2nd argument to **esym** and/or escape some of the pattern valued characters. See Section 5.2.1 Meta Characters -esym ...for an explanation and examples.

-etd(***TypeDiff*** [**,** ...] **)** inhibits
+etd(***TypeDiff*** [**,** ...] **)** re-enables
messages arising through certain specified type differences. Chapter 19. MESSAGES details the various type differences (under the heading ***TypeDiff***) and some messages are parameterized by type differences. For example, **-etd(ellipsis)** will inhibit messages reported as the result of two function types differing in that one is specified with an ellipsis and the other is not. The ***TypeDiff*** must be an identifier or of the form ***identifier/identifier***; it may not be of the form ***Type*** = ***Type***, or ***Type*** vs. ***Type*** or otherwise compound.

-etemplate(*#* [**,***#*] ... **)** inhibits
+etemplate(*#* [**,***#*] ... **)** re-enables
error message(s) *#* while expanding templates. This message suppression is in addition to other message suppression that you may have specified.

The purpose of **-etemplate** is to reduce the scrutiny of templates owing to the fact that some types and constants are unknown until expansion time. What may be a suspicious expression under normal circumstances becomes perfectly reasonable in a template situation.

Wild card characters are permitted. For example:

```
-etemplate( 7??, 8??, 17?? )
```

removes from template expansion all messages at the **700**, **800** and **1700** levels.

By default the set of messages suppressed consists of only one message, **506**. This suppression could be removed with a **+etemplate(506)**, or even **+etemplate(*)**. As this example suggests, if you attempt to re-enable a message with **+etemplate** that had not been inhibited by **-etemplate**, there will be no effect. Internally there is a set of messages associated with **etemplate** suppression. Any **-etemplate** adds to this set. Any **+etemplate** subtracts from it.

+etype(*#*,*Typename1*[,*Typename2*, ...]) re-enables
-etype(*#*,*Typename1*[,*Typename2*, ...]) 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 ***Typename1***, ***Typename2***, ... Both *#* and the ***Typename*** parameters may contain wild card characters. This option is similar to **-esym** except that it operates on the name of the symbol's type as opposed to the name of the symbol. It must be emphasized that this option applies only to ***Symbol*** parameters and not ***Name*** parameters or other kinds of parameters.

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

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

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

Also of interest ...

-w*Level* Set warning Level - See Section 5.7 Other Options

-wlib(*Level*) Set warning Level for Library - See Section 5.7 Other Options

Examples of Error Inhibition Options

```
lint -e720  p.c
```

will inhibit message **720.**

```
lint  -e????  +e1526  p.c
```

will turn off all messages except number **1526**. (Note: **-w0** is easier and just as effective as **-e????**).

```
lint  -epp  -eau  -esym(526,alpha)  p.c
```

will inhibit errors arising from pointer-pointer clashes and unsigned arguments and will suppress complaints about **alpha** not being defined.

5.3 Size and Alignment Options

This group of options allows setting the sizes and alignment of various scalars (**short**, **int**, etc.) for the target machine. The separate setting of these parameters is not normally necessary as the default settings are consistent with most compilers in your environment. Use the size options for specifying architectures other than the native architectures. For example, if you are linting for an embedded system where **int** and pointers are normally 16 bits you should specify

```
lint -si2 -sp2 ...
```

The current values of each of these size parameters can be obtained from the help screen. Try:

```
lin -sp6 ?
```

In the list below # stands for a small integer.

-sb# The number of bits in a byte is #. **-sb8** is the usual default. The number of bits in an **int** is presumed to be **sizeof(int)** times this quantity. The maximum integer is determined from this quantity by assuming a 2's complement machine. The maximum integer, in turn, is used to determine whether a constant is **int** or **long**.

-sbo# **sizeof(bool)** becomes #. The default value is 1.

-sc# **sizeof(char)** becomes #. The default value is, of course, 1. (In some rare cases the size of a **char** is something other than 1.

-slc# **sizeof(long char)** becomes #. The default is 2. This requires the flag **+flc** (enable **long char**). This type is non-standard and is employed by relatively few compilers.

-ss# **sizeof(short)** becomes #.

-si# **sizeof(int)** becomes #. For PC-lint, **-si2** is default for **LINT.EXE**. **-si4** is default for **LINT-NT.EXE**, and for **LINT-OS2.EXE**.

-sl# **sizeof(long)** becomes #.

-sll# specifies the size of **long long**. The option **-sll#** can be used to specify the size of a **long long int**. By default the size is **8**. Specifying this size also enables the flag **+fll**.

-sf# **sizeof(float)** becomes #.

-sd# **sizeof(double)** becomes #.

-sld# **sizeof(long double)** becomes #.

-sp# size of pointers becomes #. There are actually 4 different pointer sizes, **near** data, **far** data, **near** program and **far** program. This option sets all four to be a given size.

-spN# indicates that the size of **near** pointers (both of program and data) is # bytes. The PC-lint default value is 2 for **LINT.EXE**. The default is 4 for **LINT-NT.EXE**.

-spF# indicates that the size of **far** pointers (both program and data) is # bytes. The PC-lint default value is 4 for **LINT.EXE**; for **LINT-NT.EXE** the default is 6.

-spD# indicates that the size of a Data pointer (both **near** and **far**) is # bytes. This has no effect on the assumed size of program (function) pointers.

-spP# indicates that the size of a Program (function) pointer (both **near** and **far**) is # bytes. This has no effect on the assumed size of data pointers.

-spFD# indicates the size of a **far** Data pointer.

-spFP# indicates the size of a **far** Program pointer.

-spND# indicates the size of a **near** Data pointer.

-spNP# indicates the size of a **near** Program pointer.

-smp# size of Member Pointers becomes #. There are actually 3 different member pointer sizes, member data, member **near** program, and member **far** program. This option sets all three to a given size.

-smpD# indicates that the size of member Data pointers is # bytes. This has no effect on the assumed size of member program (function) pointers.

-smpP# indicates that the size of member Program (function) pointers (both **near** and **far**) is # bytes. This has no effect on the size of member data pointers.

-smpFP# indicates the size of a member **far** Program pointer.

-smpNP# indicates the size of a member **near** Program pointer.

-sw# The size of **wchar_t** is set to the indicated value. For example: **-sw4** sets wide characters to have a size of 4 bytes. Note this only affects a built-in **wchar_t** and is only useful if the built-in **wchar_t** is enabled (with **+fwc** or **+fwu**). Please see the comments for **+fwc**.

Alignment

The address at which an object may be allocated must be evenly divisible by its type's alignment. For example, if the type **int** has an alignment of 2 then each **int** must be allocated on an even byte address.

Alignment has only minimal impact on the behavior of PC-lint/FlexeLint. At this writing there are only two Elective Notes (**958** and **959**) that detect alignment irregularities. But in addition to these it is sometimes essential to get alignment correct in order to know the sizes of compound data structures.

For every size option of the form:

-s*Type#*

(except for **-sb#**, the byte size) there is an equivalent alignment option having the form:

-a*Type#*

For example, the option

-ai1

indicates that the alignment of **int** is 1 byte. An alignment of 1 means that, no restriction is placed on the alignment of types. (An alignment of 0 is undefined).

If an alignment for a type is not explicitly given by a **-a** option, an alignment is deduced from the size of the type. The deduced alignment is the largest power of 2 that evenly divides the type. Thus if the size is 4 the deduced alignment is 4 but if the size is 6 the deduced alignment is 2. This deduction is made just before the first time that alignment (and size) may be needed. An attempt to use the **-a*Type*** option (or the **-s*Type*** option) after this point is greeted with Error **75**. For example:

```
-si8        // sizeof(int) becomes 8
            // alignment of int also becomes 8
-ai1        // alignment of int is 8
-si16       // sizeof(int) becomes 16
            // alignment of int stays at 1
-sl24       // sizeof(long) is 24
            // alignment of long is 8
-al2        // alignment of long becomes 2
```

5.4 Verbosity Options

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

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

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

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

-v Turn off all verbosity messages.

+v This does not turn off verbosity messages but directs those that exist to standard error as well as to standard out.

-v*n* (where ***n*** is some integer) will print a message every ***n*** lines. This option implies **-vf**. This option will also trigger a file resumption message.

-va... Will cause a message to be printed each time there is an Attempt to open a file. This is especially useful to determine the sequence of attempts to open a file using a variety of search directories.

`-vc`... This will cause a message to be printed each time a function is called with a unique set of arguments. This is referred to as a Specific Call. See Section 10.2.2 Interfunction Value Tracking

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

`-vf` Print the names of all source Files as they are encountered. This means all headers as well as module names. Thus, `-vf` implies `-vm`. This option will indicate which headers are "Library Header Files". See Section 6.1 Library Header Files

`-vh`... At termination of processing the strong type Hierarchy be dumped in an elegant tree diagram. See the example in Section 9.6.1 The Need for a Type Hierarchy.

`-vh-`... The '`h`' verbosity flag continues to mean that the strong type hierarchy is dumped upon termination. If the '`h`' is followed immediately by a '`-`' then the hierarchy will be compressed, producing the same tree in half the lines. See Section 9.6.5 Printing the Hierarchy Tree for an example.

`-vi`... Output the names of Indirect files (`.lnt`) as they are encountered. This letter '`i`' may be combined with others.

`-vm` Print the names of Modules as they are encountered (this is the default).

`-vo`... Output Options as they are encountered whether they are inside lint comments or on the command line. The letter '`o`' may be combined with other letters.

`-vs`... The amount of Storage consumed is printed along with the verbosity message. For example, `-vsf` will report the amount of storage consumed at the start of every file processed.

`-vt`... The '`t`' flag may be added to the verbosity option. This will cause a message to be printed each time a template is to be instantiated.

`-vw`... This verbosity flag will issue a report whenever a function is to be processed with specific arguments. This is called a Specific Walk. See Section 10.2.2 Interfunction Value Tracking

`-v#`... The character '`#`' is usually used with '`f`' and will request identification numbers be printed with each file. This is used to determine whether two files are regarded as being the same.

For example:

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

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

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

5.5 Flag Options

Options beginning with **+f**, **++f**, **-f**, or **--f** introduce flags. A flag is represented internally by an integer and is considered

ON if the integer > **0**
OFF if the integer <= **0**

Default settings are either **1** if ON or **0** if OFF.

+f... turns a flag ON by setting the flag to **1**.
-f... turns a flag OFF by setting the flag to **0**.
++f... increments the flag by **1**.
--f... decrements the flag by **1**.

The latter two operations are useful in cases where you may want to turn a flag ON locally without disturbing its global setting. For example:

```
/*lint  ++flb  */
int printf();
/*lint  --flb  */
```

can be used to set the **flb** (library) flag ON for just the one declaration and, afterward, restoring the value of the flag to whatever it had been.

The current setting of all flags can be obtained from the option summary. Try the following:

```
lin -oe(temp) +fce ?
```

The file **temp** will capture the option summary showing the settings of all flags and showing, in particular, that the Continue-on-Error flag is ON.

f@m commercial **@** is a Modifier flag (default OFF).
Options: **+f@m** **-f@m** **++f@m** **--f@m**
This is a feature required by some embedded compilers that employ a syntax such as:

```
int @interrupt f() { ... }
```

the **@interrupt** serves as a modifier for the function **f** (to indicate that **f** is an interrupt handler). Normally '@' would not be allowed as a modifier. If the option **+f@m** is given then '@' can be used in the same contexts as other modifiers. There will be a warning message (**430**) but this can be suppressed with a **-e430**. The '@' will otherwise be ignored. The keyword that follows should be identified either as a macro with null value as in **-dinterrupt=** or as a reserved word using **+rw(interrupt)**.

fab ABbreviated structure flag (default OFF).
Options: **+fab -fab ++fab --fab**
if this flag is ON, structure references may be abbreviated. Thus, instead of **s.a.b**, if it would cause no ambiguity, you may use **s.b**. Few compilers support this feature.

fai Arguments pointed to get Initialized flag (default ON).
Options: **+fai -fai ++fai --fai**
When an argument is passed to a function in the form of the address of a scalar and if the receiving parameter is not declared as **const** pointer, then it is assumed by default that the scalar takes on new values and that we do not know what those values are. Thus, in the following:

```
void f( int ** );
   void g()
       {
       int *p = NULL;
       f( &p );
       *p = 0;     // OK, no warning.
       }
```

we do not warn of the possibility of the use of a NULL pointer because our past knowledge is wiped away by the presumed initialization afforded by the function **f**. However, if the flag is turned OFF (using **-fai**), then we will warn of the possibility of the use of a NULL pointer.

fan ANonymous union flag (default OFF for C, ON for C++).
Options: **+fan -fan ++fan --fan**
If this flag is ON, anonymous unions are supported. Anonymous unions appear within structures and have no name within the structure so that they must be accessed using an abbreviated notation. For example:

```
struct abc
        {
        int n;
        union { int ui; float uf; };
        } s;
     ... s.ui ...
```

In this way a reference to one of the union members (**s.ui** or **s.uf**) is made as simply as a reference to a member of the structure (**s.n**).

This is a feature of the Microsoft compiler and is also in C++.

fas Anonymous struct flag (default OFF).
Options: **+fas -fas ++fas --fa**
If this flag is ON (by default it is OFF), anonymous struct's are supported. Anonymous struct's are similar to anonymous union's. This is, if a struct has no tag and no declarator as in:

```
class X
    {
    struct { int a; int b; };
    } x;
```

then references to the members of the inner struct are as if they are members of the containing struct. Thus **x.a** refers to member **a** within the unnamed struct within class **X**. Some compilers (most notably MSC) support this construct, but it is non-standard. When the **-cmsc** option is given, this flag is set ON.

fat Parse **.net** ATtributes flag (default ON).
Options: **+fat -fat ++fat --fat**
Dot net (**.net**) attributes are contained within square brackets. E.g.

```
[propget, id(1)] void f( [out] int *p );
```

The square brackets and information contained therein are non standard extensions to the C/C++ standards supported by the Microsoft Visual C 7.00 and later compilers. Remarkably this doesn't appear to interfere (or be ambiguous) with other uses of square brackets within the language. For this reason the flag is normally ON. To turn off such processing use **-fat**

fba Bit Addressability flag (default OFF).
Options: **+fba -fba ++fba --fba**
If this flag is ON (by default it is OFF), an individual bit of an **int** (or any integral) can be specified using the notation:

```
a.integer-constant
```

where **a** is an expression representing the integral and ***integer-constant*** is an integer constant. The construct is treated as a bit field of length 1.

For example, the following code:

```
/*lint +fba  Turn on Bit Addressability */
int n;
n.2 = 1;
n.4 = 0;
...
if( n.2 || n.4 ) ...
```

will set the 2nd bit of **n** to 1 and the 4th bit to 0. Later it tests those bits.

This syntax is not standard C/C++ but does represent a common convention used by compilers for embedded systems.

fbc Binary Constant flag (default OFF).
Options: **+fbc -fbc ++fbc --fbc**
If this flag is ON, binary constants having the form **0{bB}{01}+** are permitted. For example:

```
0B101010101 == 0x1555
0b1111 == 0xF
```

This is not an ANSI/ISO construct and few compilers actually support this feature.

fbo Boolean flag (default ON).
Options: **+fbo -fbo ++fbo --fbo**
If this flag is ON, keywords **bool**, **true**, and **false** are activated at the start of every C++ module.

fbu Bit fields are **unsigned** flag (default OFF).
Options: **+fbu -fbu ++fbu --fbu**
When this flag is ON, bit fields will be considered **unsigned** even though a nominal **int** is used in its specification. For example:

```
struct S { int b:1; };
```

has a one-bit bit field **b** whose values normally range from -1 to 0. It will receive an Informational message (**806**) warning about the unusual character of the declaration. However some compilers, treat this as ranging from 0 to 1. i.e., they treat the bit field as **unsigned**.

The preferred programmer response is to declare the bit field **unsigned**, as this is portable. If you don't want to do this for some reason, you may set this flag ON, to match the behavior of your compiler.

fcd **cdecl** is significant flag (default OFF).

Options: **+fcd** **-fcd** **++fcd** **--fcd**
If this flag is ON, a distinction is made between functions declared explicitly with the **cdecl** modifier (meaningful to users of PC-lint) and functions declared without this modifier. When generating prototypes (**-od**), **cdecl** is added for all functions declared with this modifier.

fce Continue on Error flag (default OFF).
Options: **+fce** **-fce** **++fce** **--fce**
If a **#error** directive is encountered, processing will normally terminate. If this flag is ON, the **#error** line is printed and processing will continue.

fcp C++ flag (default OFF).
Options: **+fcp** **-fcp** **++fcp** **--fcp**
When this flag is ON, all subsequent modules will be processed as C++ modules, not just the ones having a distinguished extension (by default "**.cpp**" and "**.cxx**").

fct Create Tag flag (default OFF).
Options: **+fct** **-fct** **++fct** **--fct**
Any **struct**, **union**, **enum**, or **class** that has no name following the keyword is considered untagged. If this flag is ON (the default is OFF), tags will be created where they do not otherwise exist. The created tag will consist of a compound name containing the file name and line number where the structure is defined. There are a number of uses for such a flag. For one, it forces untagged structs to be considered different types if they are defined in different files. Also, some subtle problems can occur when attempting to reconstruct **lob** information that contains untagged structures. The simple remedy in that case is to set this flag.

fcu **char** is **unsigned** flag (default OFF).
Options: **+fcu** **-fcu** **++fcu** **--fcu**
If this flag is ON, plain **char** declarations are assumed to be **unsigned**. This is useful for compilers, which, by default, treat **char** as **unsigned**. Note that this treatment is specifically allowed by the ANSI/ISO standard. That is, whether **char** is **unsigned** or **signed** is up to the implementation. See also the **fdc** flag.

fda Double quotes to Angle brackets flag (default OFF).
Options: **+fda** **-fda** **++fda** **--fda**
This flag, which is normally OFF, affects include processing. The effect is that when attempting to open a file **a.h**:

```
#include "a.h"
```

Lint will behave as if you had included the file with angle brackets instead of double-quotes. Note that the **+fdi** option has no effect when this flag is ON. This option was added to support users of the ARM compiler (whose **-fd** option has the same effect on include processing).

fdc Distinguish plain **char** flag (default ON).
Options: **+fdc -fdc ++fdc --fdc**
This option in effect says that for the purpose of function overloading there are three distinct **char** types: **char**, **signed char** and **unsigned char**. This is the C++ standard **[10]** and is set automatically if you turn on the ANSI/ISO flag (**-A**). If your compiler does not support all three, then the option **-fdc** must be given in the compiler options file. You can tell if your compiler needs it by noting whether, for example, both **char** and **signed char** are overloaded for the same functions. Look in **iostream.h** for example.

fdd Dimension by Default flag (default ON)
Options: **+fdd -fdd ++fdd --fdd**
If this flag is ON then strong types (with the 'J' flag) will be considered equivalent to 'Jd'. The resulting behavior will be equivalent to treating the type as a physical dimension such as meters or seconds. See Section 9.4 Multiplication and Division of Strong Types. This departs from the earlier interpretation (pre Version 9) of treating strong types with the J property as being anti-dimensional ('Ja'). To obtain the old behavior as the default interpretation, you may turn this flag OFF using the option **-fdd**.

fdh Dot-H flag (default OFF).
Options: **+fdh -fdh ++fdh --fdh**
When the Dot-H flag is ON (**+fdh**) and if an extension-less header is seen, then a "**.h**" is appended. In prior versions, an open was attempted on the **.h** extended name but not on the original name. This proved to be troublesome when one was mixing libraries. In the current version, an attempt is made to open the file first with the **.h** extension and, that failing, open the file using the original name.

To obtain the prior behavior you may increment the Dot-H flag to a value of **2**. Thus:

```
+fdh ++fdh
```

will achieve the earlier behavior.

fdi Directory of Including file flag (default varies).
Options: **+fdi -fdi ++fdi --fdi**
For Unix systems the default is ON. For PC-lint and for other systems the default is OFF. If this flag is ON, the search for **#include** files will start with the directory of the including file (in the double quote case) rather than with the current directory. This is the standard Unix convention and is also used by the Microsoft compiler. For example:

```
#include "alpha.h"
```

begins the search for file **alpha.h** in the current directory if the **fdi** flag is OFF, or in the directory of the file that contains the **#include** statement if the **fdi** flag is ON. This normally won't make any difference unless you are linting a file in some other directory as in:

```
lint  source\alpha.c
```

If **alpha.c** contains the above **#include** line and if **alpha.h** also lies in directory **source** you need to use the **+fdi** option.

fdl pointer Difference is **long** flag (default OFF).
Options: **+fdl -fdl ++fdl --fdl**
This flag specifies that the difference between two pointers is typed **long**. Otherwise the difference is typed **int**. This flag is automatically adjusted upon encountering a **typedef** for **ptrdiff_t**.

If the value of the flag is 2, then pointer differences are assumed to be **long long.** This can occur through the pair of options:

```
+fdl    ++fdl
```

fdr Deduce Return mode flag (default OFF).
Options: **+fdr -fdr ++fdr --fdr**
With the introduction and general acceptance of the **void** type this option is almost obsolete. The return mode of a function has to do with whether the function does, or does not, return a value. This flag only affects function definitions and declarations that do not have an explicit return type. This can be a very valuable option for older C programs. If the flag is ON, **return** statements are examined to determine the return mode of such a function. If the flag is OFF, such a function is assumed to return an **int**. With the flag OFF we are adhering strictly to ANSI/ISO.

feb allow Enumerations as Bit fields flag (default ON).
Options: **+feb -feb ++feb --feb**
The following is technically illegal but is allowed by some compilers.

```
enum color { red, green, blue };
     struct {
        enum color x : 2;
        enum color y : 2;
            };
```

The base type of a bit field should be an **int** (signed or unsigned). If this flag is ON, no complaint is issued at declaration time nor at assignment time when assigning a compatible **enum**.

fem Early Modifiers flag (default OFF).
Options: **+fem -fem ++fem --fem**
Microsoft modifiers such as **far**, **_near**, **__huge**, **_pascal**, etc. modify the declarator to its immediate right and so strictly speaking

```
pascal int f();
```

is not valid and will yield error **140**. It should be:

```
int pascal f();
```

Some compilers (such as Metaware) accept both forms and some of their header files use the former version. To support this alternative form (i.e. to allow early modifiers) use the **+fem** flag. Alternatively you may want to have early modifiers flagged only in your own code. To suppress the message only within library headers use **-elib(140)** instead of setting this flag.

fep Extended Pre-processor variable flag (default OFF).
Options: **+fep -fep ++fep --fep**
Normally, enumeration constants and const variables that serve as constants are ignored when computing **#if** directives. Although the standard does not allow such variables, some compilers do. If this flag is ON they are permitted. Thus:

```
//lint +fep
const int N = 2;
enum { M = 3 };
#if N < M  /* allowed only if +fep is ON */
#endif
```

fet Explicit Throw flag (default OFF).
Options: **+fet -fet ++fet --fet**
If the flag is OFF then the absence of an exception specification (the throw list for a function) is treated as a declaration that the function can throw any exception. This is standard C++. If the flag is ON, however, the function is assumed to throw no exception. In effect, the flag says that any exception thrown must be explicitly given. Consider

```
double sqrt( double x ) throw( overflow );
double abs( double x );
double f( double x )
    {
    return sqrt( abs(x) );
    }
```

In this example, **sqrt()** has an exception specification that indicates that it throws only one exception (**overflow**) and no others. The functions **abs()** and **f()**, on the other hand, have no exception specification, and are, therefore, assumed to potentially throw all exceptions. With the Explicit Throw flag OFF you will receive no warning. With the flag ON (with a **+fet**), you will receive Warning **1550** that exception **overflow** is not on the throw list of function **f()**.

The advantage of turning this flag ON is that the programmer can obtain better control of his exception specifications and can keep them from propagating too far up the call stack. This style of analysis is very similar to that employed quite successfully by Java.

The disadvantage, however, is that by adding an exception specification you are saying that the function throws no exception other than those listed. If a library function throws an undeclared exception (such as **abs()** above) you will get the dreaded **unexpected()** function call. See Scott Meyers "More Effective C++", Item 14.

Can you have the best of both worlds? Through the magic of macros it would appear that you can. For example, you can define a macro **Throw** as follows:

```
#ifdef _lint
    #define Throw( x ) throw(x)
#else
    #define Throw( x )
#endif
```

When linting you would turn on the **+fet** flag. You would then use the **Throw** macro for all your exception specifications that PC-lint/FlexeLint is warning you to add. These specifications will not be seen by the compiler and therefore will not get you into trouble.

Unfortunately, you will soon discover, that **Throw** doesn't handle the multiple argument case. Clearly you can define a series of separate macros, **Throw2**, **Throw3**, etc. for different argument counts. But you can also define a multiple argument macro **Throws** as follows:

```
#define Throws(X) throw X
```

Unfortunately, this requires an extra set of parentheses when you use it as in:

```
Throws((overflow,underflow))
```

But this is not necessarily a bad thing since it will alert the reader of the code that these are not seen by the compiler.

ffb **for** loop creates separate Block flag (default ON).
Options: **+ffb** **-ffb** **++ffb** **--ffb**
The C++ standard designates that variables declared within **for** clauses are not visible outside the scope of the **for** loop. For example, in the following code, **i** can't be used outside the **for** loop.

```
for( int i = 0; i < 10; i++ )
        {
        // ...
        }
```

```
// can't use i here.
```

Some compilers still adhere to an earlier practice in which variables so declared are placed in the nearest encompassing block.

By default, this flag is ON indicating that the standard is supported. If your compiler follows a prior standard you may want to turn this OFF with the option

```
-ffb
```

ffc Function takes Custody flag (default ON).
Options: **+ffc** **-ffc** **++ffc** **--ffc**
This flag is normally ON. It signifies that all non-library functions will automatically assume custody of a pointer through any non-const pointer parameter. Turning this flag OFF (with a **-ffc**) will mean that a given function will not take custody of a pointer unless explicitly directed to do so via a custodial semantic for that function and argument.

```
void f( int * );         // f takes custody
void g( const int * );   // but g does not
//lint --ffc                 turn the flag off
void h( int * );         // h will not take custody
//lint ++ffc                 restore the flag
```

See option **-sem**. See also message **429**.

ffd Float to Double flag (default OFF).
Options: **+ffd** **-ffd** **++ffd** **--ffd**
If this flag is ON, **float** expressions are automatically promoted to **double** when being used in an arithmetic expression (just as **char** is promoted to **int**). Automatic **float** promotion is K&R C but not ANSI/ISO C.

fff Fold Filenames flag (default ON).
Options: **+fff** **-fff** **++fff** **--fff**
On some operating systems, filenames are by default folded to either upper case (VMS and MVS) or to lower case (DOS, OS/2, Macintosh). This aids in filename identification and also provides for a more uniform presentation. On other systems, such as Windows. filenames are not folded for the purpose of saving and later displaying the name but they are folded for the purpose of name comparison. Thus "**a.h**" will compare equal to "**A.H**". Also both "**.CPP**" and "**.cpp**" are extensions which by default will be regarded as C++ extensions.

If such folding is unwanted, you may employ the **-fff** option to reset the Fold Filenames flag.

For systems that do not fold filenames (E.g. Unix and Linux) this flag has no effect.

ffn Full (file) Name flag (default OFF).
Options: **+ffn -ffn ++ffn --ffn**
When ON filenames reported in error messages are full path names. This can assist editors in locating the correct position within files when default directories may be different than during the linting.

ffo Flush Output files flag (default ON).
Options: **+ffo -ffo ++ffo --ffo**
When ON, the **fflush()** function is called after each message. Otherwise messages are buffered. If there are many messages it is slightly faster to buffer.

fhc The Hash Created names flag (default ON)
Options: **+fhc -fhc ++fhc --fhc**
Created names (i.e. those created to represent the unnamed namespace, static storage, and created tags (see **+fct**)) are made in such a way as to be unique for a particular module no matter how long the pathname. To avoid the possibility of having disconcertingly long names the full path name can be hashed. The resulting hash number is then concatenated with the root of the module name to obtain a virtually guaranteed unique name.

This is the default behavior and the Hash-Created-names flag is said to be ON by default. To turn off the hashing of created names use the option **-fhc**.

fhd strong Hierarchy Down flag (default ON).
Options: **+fhd -fhd ++fhd --fhd**
This flag is ON by default. The strong-Hierarchy-Down flag refers to assignments between strong types related to each other via the strong type hierarchy. Normally you may freely assign up and down the hierarchy without drawing a warning. With this flag set OFF a warning will be issued whenever you assign down the hierarchy. For example:

```
typedef int X;
typedef X Y;
```

Then an assignment from an object of type **X** to a variable of type **Y** will draw a warning if the flag is turned off. See also **-father()** and **-parent()** in Section 5.7 Other Options.

fhg Hierarchy Graphics flag (default varies).
Options: **+fhg -fhg ++fhg --fhg**
For PC-lint the default is ON; for FlexeLint the default is OFF. If this flag is ON, IBM graphics characters are used to display a type hierarchy tree. See Section 9.6.5 Printing the Hierarchy Tree

fhs Hierarchy of Strong types flag (default ON).
Options: **+fhs -fhs ++fhs --fhs**
If this flag is ON (it is by default) strong types are considered to form a hierarchy based on **typedef** statements. See Section 9.6.2 The Natural Type Hierarchy and Section 9.6.3 Adding to the Natural Hierarchy.

fhx Hierarchy of strong indeXes flag (default ON).
Options: **+fhx -fhx ++fhx --fhx**
If this flag is ON (it is by default) strong index types are related via the type hierarchy. See Chapter 9. STRONG TYPES. See also the **+fhs** flag.

fie Integer model for Enum flag (default OFF).
Options: **+fie -fie ++fie --fie**
If this flag is ON, a loose model for enumerations is used (loose model means that enumerations are regarded semantically as integers). By default, a strict model is used wherein a variable of some enumerated type, if it is to be assigned a value, must be assigned a compatible enumerated value and an attempt to use an enumeration as an **int** is greeted with a (suppressible) warning **641**. An important exception is an **enum** that has no tag and no variable. Thus

```
enum {false,true};
```

is assumed to define two integer constants and is always integer model.

fii Inhibit Inference flag (default OFF).
Options: **+fii -fii ++fii --fii**
The purpose is to suppress inference formation during Specific Walks and during the evaluation of expressions involving nul-terminated strings. These inferences were prone to error and were resulting in undeserved messages.

Owing to steady progress in the accuracy of making such inferences, the flag has been made default OFF. To obtain the prior behavior (i.e. as it was in version 8.00L) turn the flag on with **+fii**.

fil Indentation check on Labels flag (default OFF).
Options: **+fil -fil ++fil --fil**
Normally no indentation check is done on labels because frequently they are positioned far to the left of a listing in a position of prominence and easy visibility. If you want labels checked, turn this flag ON. See Section 13.3 Indentation Checking

fim Include Multiple flag (default ON).
Options: **+fim -fim ++fim --fim**
With this flag ON, the **-i** option may specify multiple include directories (like the INCLUDE environment variable). For example, for DOS,

```
-iC:\compiler\include;C:\myinclude
```

will have the same effect as:

```
-iC:\compiler\include
-iC:\myinclude
```

fiq Ignore default Qualifier flag (default OFF).
Options: **+fiq -fiq ++fiq --fiq**
If this flag is ON, then an explicit memory model qualifier is ignored if it matches the memory model selected using the **-m** option. For example, if this flag is ON then, in Large memory model (**-ml**), the following two declarations:

```
extern char far *p;
extern char *p;
```

are not distinguishable. By default (OFF), they are distinguishable and will raise a diagnostic. Normally you would want this flag OFF so that you can be aware that your declarations may not be perfectly consistent. However, some compiler and/or library header files are inconsistent in their use of these qualifiers and it is necessary to set this flag just to be able to process these header files.

fis Integral constants are Signed flag (default OFF).
Options: **+fis -fis ++fis --fis**
If this flag is ON, integer constants are typed **int** or **long**, never **unsigned** or **unsigned long**. For example, by the rules of ANSI/ISO (rules described in Section 13.11 Size of Scalars), **0xFFFF** is considered **unsigned** if **int** are 16 bits. However, some older compilers regard all integral constants as signed. To mimic these use **+fis**.

fiw Initialization is considered a Write flag (default ON).
Options: **+fiw -fiw ++fiw --fiw**
This flag is normally ON. When this flag is ON, any initialization is considered a Write to the variable being initialized (unless inhibited in some other way, see the **fiz** flag below). Two successive Writes to the same variable are flagged with Info **838**. Thus:

```
int n = 3;
n = 6;          // Info 838
```

is normally greeted with message **838**. If the flag is turned off (with a **-fiw**), the message would not be issued because the assignment of **6** to **n** would be considered the first Write. A subsequent Write without a Read would be unaffected by the flag and generate Info **838**. See also Warning **438**, Info **838** and the **-fiz** flag below.

fiz Initialization by Zero is considered a Write flag (default ON).
Options: **+fiz -fiz ++fiz --fiz**
This flag is normally ON. When this flag is ON, an initialization by 0 is considered a Write to the variable being initialized. Two successive Writes (without an intervening Read) to the same variable are flagged with Info **838**. Thus in the code:

```
int n = 0;
n = 6;          // Info 838 depends on fiz flag
n = 16;         // Info 838 always
```

The assignment of **6** to **n** is normally greeted with message **838**. If the flag is turned off (with a **-fiz**), the message would not be issued because the assignment to **6** would be considered the first Write. The subsequent assignment of **16** is flagged with Info **838**.

See also messages **438**, **838** and the **-fiw** flag above.

fjm JM controls the Multiplier group flag (default ON)
Options: **+fjm** **-fjm** **++fjm** **--fjm**
This flag pertains to the **-strong** option. It is normally ON. When turned OFF with the **-fjm** the programmer can achieve backward compatibility with versions prior to Version 9.00. This is described below.

When defining a strong type the programmer may elect to have messages issued when objects of that type are assigned (**A**), extracted (**X**) or joined (**J**). The '**J**' refers to when an operand having a strong type is combined via binary operators with other operands. Starting with Version 9.00 these operators fall into four categories:

- **e** equality operators
- **r** relational operators
- **m** multiplicative operators
- **o** other operators

A suboption of the form **Jo** requests that all Joins other that '**o**' joins be checked. The '**o**' is called a softener.

In versions prior to 9.00 there were only three categories. There was no multiplicative group and so a suboption of the form '**Jo**' implied no checking of multiplicative operations. To achieve the same effect with Version 9.00 you would be required to use the suboption '**Jmo**'.

To ease the transition to Version 9.00 you may elect to turn this flag off using the option **-fjm**. But bear in mind that if you are using strong types you will not want to set this flag OFF permanently as then you will miss out on the rich set of dimensionality checks that strong typing now provides through the multiplicative operators.

fkp K&R Preprocessor flag (default OFF).
Options: **+fkp** **-fkp** **++fkp** **--fkp**
A number of preprocessor facilities are allowed by the ANSI/ISO standards, which are not allowed in K&R C. These include blanks and tabs preceding the initial # sign. Setting this flag causes strict adherence to the K&R preprocessor specification.

flb LiBrary flag (default OFF).
Options: **+flb** **-flb** **++flb** **--flb**
If ON, code is treated as belonging to a library header (See Section 6.1 Library Header Files). That is, declared objects don't have to be used or defined and messages specified by **-elib** (See Section 5.2 Error Inhibition Options) are suppressed. The flag is automatically turned OFF at the end of a module. This flag has been largely superseded by the notion of "Library Header Files" (See Section 6.1 Library Header Files). It still has its uses though. For example, the output of PC-lint/FlexeLint in preprocess mode (i.e. using the **-p** option) will contain Lint comments bearing **++flb** before and **--flb** after the positions at which library headers were included.

This setting and unsetting of this flag is kept independently of the notion of library header (or module). Librariness of code is determined by an OR of this flag and the Librariness of the file. In this way, sections of a file can be library while the rest is not.

flc **long char** flag (default OFF).
Options: **+flc** **-flc** **++flc** **--flc**
If the long-char flag is ON (option **+flc**) then **long char** is a permitted type, which results in an integral quantity nominally different from **char**, **short**, **int** or **long int**. Some compilers use this type to specify wide characters (the ANSI/ISO type **wchar_t**). The size of the wide character can be specified with the option **-sw#**.

fld Label Designator flag (**+fld**)
Options: **+fld** **-fld** **++fld** **--fld**
This flag allows the use of the non-standard feature called a label designator. An earlier version of the GCC compiler offered a feature called a label designator. This permitted the programmer to designate the position of initialization of a **struct** via the sequence: '***identifier*:**'. This would designate the field named by the ***identifier***.

For example:

```
struct A { int a, b, c; } x = { b: 33 };
```

will initialize to **33** the field named '**b**'.

This particular form is obsolete even for GCC as it has been replaced by the C99 standard of

'**.**' ***identifier*** '**=**'

flf process Library Function flag (default OFF).
Options: **+flf** **-flf** **++flf** **--flf**
If this flag is ON, library function definitions will be fully processed. It is normally OFF to save a little time in processing large C++ header files. Processing library function definitions means that each in-line function definition will be parsed looking for the usual suspicious coding flaws. Normally the programmer is not interested in examining flaws in

the library and so would prefer to leave this flag OFF. Occasionally someone is working on a library and would prefer to have all such functions closely examined.

fll **long long** flag (default OFF).
Options: **+fll** **-fll** **++fll** **--fll**
If the long-long flag is ON (option **+fll**) then **long long int** (or just **long long**) is a permitted type, which results in an integral quantity nominally different and usually longer than a **long int**. The size of a **long long int** can be specified with the option **-sll#**. If the **long-long** flag is not set, then you will be warned (Info **799**) if an integral constant exceeds the size of a **long**.

flm Lock Message format flag (default OFF).
Options: **+flm** **-flm** **++flm** **--flm**
This flag can be used by GUI front ends which depend on a particular format for error messages. If this flag is ON, the Message Presentation Options (See Section 5.6. Message Presentation Options) are frozen. That is, subsequent **-h**, **-width**, and the various **-format** options are ignored. Also ignored is the **-os** option. The **-os** option (See Section 5.7 Other Options) designates which file will receive error messages.

fln **#line** directives flag (default ON).
Options: **+fln** **-fln** **++fln** **--fln**
By default, **#line** directives affect the location information within error messages. The option **-fln** may be used to ignore **#line** directives. See Section 15.3.1 #line and # for an example.

Some systems utilize # as a synonym for **#line**. To support this usage, the **fln** flag must be incremented to 2 or more. This can be done with the option **++fln** (the initial value of 1 is incremented to obtain 2).

fmc Macro Concatenation flag (default OFF).
Options: **+fmc** **-fmc** **++fmc** **--fmc**
If the flag is ON, a token immediately following a macro with parentheses, will, in effect, be pasted on to the end of the last token of the macro replacement. For example, the code

```
#define A() a
int A()x;
```

will normally be greeted with an error according to the ANSI/ISO standards because this is equivalent to:

```
int a x;
```

However if the Macro Concatenation flag is turned on (using the option **+fmc**) the two names are in effect pasted together to produce the equivalent of:

```
int ax;
```

Prior to the ANSI/ISO standard the only way to perform a concatenation of this kind was through the device described above. Now the approved mechanism is through the ## operator. This option is a means of supporting older programs that are still employing the earlier technique.

fmd Multiple Definitions flag (default OFF).
Options: **+fmd -fmd ++fmd --fmd**
Some compilers allow multiple definitions of data items provided they are not accompanied by an initializer. These are referred to in ANSI/ISO C as tentative definitions. For example, in the sequence:

```
int n;
int n = 3;
int n;
int n = 3;
```

some compilers would consider only the last declaration as erroneous. If the **fmd** flag is ON, only the last declaration draws the "Previously defined" error message (number **14**). Multiple definitions of functions are always reported.

fna Allow **operator new[]** flag (default ON).
Options: **+fna -fna ++fna --fna**
In C++ it is possible to override **operator new** by writing your own member function with this name. **operator new[]** is a similar function that you can override that will be used to allocate space for arrays. But not all compilers support this new feature. With **+fna** ON, the New Array facility will be activated.

fnc Nested Comments flag (default OFF).
Options: **+fnc -fnc ++fnc --fnc**
If this flag is ON, comments may be nested. This allows PC-lint/FlexeLint to process files in which code has been 'commented out'. Commenting out code should not be considered good practice, however. Code should be disabled by using a preprocessor conditional as it avoids the quoted star-slash problem and it automatically assigns a condition to the re-enabling of the code.

fnn New can return Null flag (default OFF).
Options: **+fnn -fnn ++fnn --fnn**
Turning this flag ON yields the old style **operator new**. That is, new may return NULL and does not throw an exception.

According to Standard C++, there are two built-in functions supporting **operator new**:

```
void *operator new( size_t ) throw( std::bad_alloc );
```

```
void *operator new[]( size_t ) throw( std::bad_alloc );
```

Rather than return NULL when there is no more allocatable space, these functions throw an exception as shown.

However, earlier versions of the language, especially before there were exceptions, returned NULL when storage was exhausted. To support this older convention, this flag was created.

fnr NULL can be Returned flag (default OFF)
Options: **+fnr -fnr ++fnr --fnr**
This flag is normally OFF. When this flag is ON, then all functions that return pointers and have no other return semantic are assumed to return pointers that could possibly be NULL. For example:

```
//lint +fnr     null can be returned by functions
int *f();
void g()
    {
    int *p = f();       // p could be NULL
    if( !p ) return;    // avoid a diagnostic
    *p = 0;             // OK now to use p.
    }
```

fns Nested Struct flag (default ON).
Options: **+fns -fns ++fns --fns**
This flag governs the placing of nested classes. A nested class is a **class**, **struct** or **union** defined within another **class, struct** or **union**. A C compiler will ordinarily place a nested class in the same namespace that holds the containing class. A C++ compiler will place a nested class within the namespace of the containing class. But consider the following:

```
extern "C"
    {
    struct A
        {
        struct B
            {
            int b;
            };
        int a;
        };
    }
```

Should this (clearly C++) code nest the **B** within the **A** so that **A::B** will reference the nested **struct** or should it follow C conventions because of the **extern** "**C**" in which case **B** alone will serve as a reference. Our initial approach was to take the latter path but

this proved inconsistent with other C++ compilers so our current implementation is to nest the **B** within **A**. But that decision can be overridden as follows.

If the nested struct flag is turned OFF (default is ON) nested struct's within the scope of an **extern "C"** will not be nested.

To force the nesting of struct's within a C module you may set the value of the flag to **2** using the **++fns** option. This might be advisable to match the behavior of a C compiler or to avoid incompatibility errors between data structures in C and the same data structures referenced in C++.

fnt Nested Tag flag (default ON).
Options: **+fnt -fnt ++fnt --fnt**
A Nested Tag is a tag (**class**, **struct**, **union** or **enum** name) defined within a class. These are not normally visible outside the class unless qualified. With this flag ON, and if a context clearly indicates that a tag is required (such as before a scope operator) then a special search is made of the table of all tags. Appropriate warnings are issued if the tag is found in areas that are not truly visible.

fod Output Declared objects flag (default OFF).
Options: **+fod -fod ++fod --fod**
This flag has an effect only when a Lint Object Module (See option **-oo** in Section 5.7 Other Options) is being produced. Normally, objects declared but not referenced are not placed in the output. With this flag ON, all objects declared are placed there. This has the disadvantage of making the object modules much larger than they need to be. It has the advantage that all declared objects will be cross-checked.

fol Output Library objects flag (default OFF).
Options: **+fol -fol ++fol --fol**
This flag has an effect only when a Lint Object Module (See option **-oo** in Section 5.7 Other Options) is being produced. Normally, objects declared when the library flag is set (see **+flb** in Section 5.5 Flag Options and/or **-library** in Section 5.7 Other Options) are not placed in the output. With this flag ON, all library objects are placed in the output module. It is not usually necessary to set this flag ON when creating a Lint Object Module that describes a library. See Section 6.1 Library Header Files

fpa PAuse flag (default OFF).
Options: **+fpa -fpa ++fpa --fpa**
When this flag is ON, PC-lint/FlexeLint will pause just before exiting (after all messages are produced), and request input through **stdin** after prompting on **stderr**. Hitting Return (i.e., Enter) should be enough to finally terminate. This option could be useful in a windowing system gone 'berserk' where the output flashes momentarily upon the screen before its window is closed. This option should keep the window open. CAUTION: This option is recommended only as a trouble shooting option or as a stop gap measure. Some environments require the launched program to terminate or they themselves lock up.

fpc Pointer Casts retain **lvalue** flag (default OFF).
Options: **+fpc -fpc ++fpc --fpc**
This flag can be used to legitimize a non-standard practice, which is rife in the C community. For example if you wanted to add 1 (1 byte not 1 **int**) to an **int** pointer (**pi**) then you could write:

```
(*(char **)&pi)++;
```

which is a lot of effort and confusing. You could write:

```
((char *)pi)++;
```

This is non-standard because the cast removes the lvalue property from **pi** and hence it can no longer be incremented. For this reason it will draw a diagnostic from PC-lint/ FlexeLint even though many (if not most) compilers accept it. If you choose the second alternative you should turn ON the **fpc** flag to suppress the message.

fpd Scalars, Pointer sizes Differ flag (default OFF).
Options: **+fpd -fpd ++fpd --fpd**
If this flag is ON, pointers to one type are considered to have different sizes than those to a different type.

fpm Precision is limited to the Maximum of its arguments flag (default OFF).
Options: **+fpm -fpm ++fpm --fpm**
This is used to suppress certain kinds of Loss of Precision messages (**734**). In particular, if multiplication or left shifting is used in an expression involving **char** (or **short** where **short** is smaller than **int**) an unwanted loss of precision message may occur. For example, if **ch** is a **char** then:

```
ch = ch * ch
```

would normally result in a Loss of Precision. This is suppressed when **+fpm** is set. This flag is automatically (and temporarily) set for operators **<<=** and ***=**.

For example

```
ch <<= 1
```

is not greeted with Message **734**.

fpn Pointer parameter may be NULL flag (default OFF).
Options: **+fpn -fpn ++fpn --fpn**
If this flag is set ON, all pointer parameters are assumed to be possibly NULL and a diagnostic will be issued if a pointer parameter is used without testing for NULL. For example:

```
void f( char *p, char *q )
    {
    *p = 3;     // warning only if +fpn
    q++;        // warning only if +fpn
    }

void g( char *p, char *q )
    {
    if( p && q )
        {
        *p = 3;     // no warning
        q++;        // no warning
        }
    }
```

For more information about this interesting test see Section 10.2 Value Tracking.

fps Parameters within Strings flag (default OFF).
Options: **+fps -fps ++fps --fps**
This flag, when set ON, allows macro parameters to be substituted within strings as in:

```
#define printi(n) printf( "n = %d\n", n )
```

which prints both the name and the value of the parameter passed to the macro **printi**. This depends on the substitution of a macro parameter within a string constant and had been supported by many compilers but is now expressly forbidden by ANSI/ISO C. There are other ways to accomplish this task, such as the # stringize operator. See also Warning **607** in Section 19.4 C Warning Messages.

fqb The Qualification Before type flag (default ON)
Options: **+fqb -fqb ++fqb --fqb**
This flag is normally ON and in conjunction with Elective Note **963** can report on declarations in which **const** and **volatile** qualifiers do not follow a consistent pattern as to whether they appear before or after types in a ***type*** specifier. By default Note **963** will report whenever a qualifier follows a type. If the **fqb** flag is turned OFF (e.g. **-fqb**) the qualifier is expected to follow the type. For example:

```
int const x;    // by default no message
//lint +e963       turn on message 963
int const y;    // msg 963: qualifier follows type
const int z;    // no msg
//lint -fqh        reverse the message
int const a;    // no msg
const int b;    // msg 963: qualifier precedes type
```

Saks **[36]** and Vandevoorde and Josuttis **[32, section 1.4]** provide supporting evidence that not only is a convention useful, but that the better convention is the one rendered with **`-fqb`**.

`frb` Read Binary flag (default OFF).
Options: **`+frb -frb ++frb --frb`**
When this flag is ON, all files opened with **`fopen`** on input are given a mode of **`rb`** rather than **`r`**. This is to resolve an obscure problem that can arise with some editor/compiler combinations. On a system (such as MS-DOS) that uses **`CR-LF`** to separate lines, some editors do not insert a **`CR`** between lines (to save space) and some run-time libraries will not stop (with **`fgets`**) on just an **`LF`** unless read with **`rb`**. Using this option will handle the situation.

`frl` Reference Location information in messages flag (default ON).
Options: **`+frl -frl ++frl --frl`**
Reference location information is a clause that is appended to some messages (usually those associated with value tracking), which is of the form:

```
[Reference:  File ... : line ... , ... ]
```

This trailer can be eliminated by turning this flag OFF.

`frn` Treat carriage Return as Newline flag (default OFF).
Options: **`+frn -frn ++frn --frn`**
If this flag is ON, carriage return characters (**`0x0D`**) in the source input not followed by a Newline (**`0x0a`**) are treated as Newline characters (i.e., as line breaks). This is necessary to process Macintosh style source code on the PC or Unix or their derivatives. With this flag ON all three conventions (**`NL`** alone, **`CR`** alone, and **`CR NL`** in combination) are taken to be a Newline so that you may mix header files.

`fsa` Structure Assignment flag (default ON).
Options: **`+fsa -fsa ++fsa --fsa`**
If this flag is ON, structure assignment is assumed to be valid. Functions, actual arguments and parameters may be typed **`struct`** or **`union`** and such objects are allowed to be used in assignment.

`fsc` String constants are **`const char`** flag (default OFF).
Options: **`+fsc -fsc ++fsc --fsc`**
When this flag is ON, string constants are considered pointers to **`const char`**. For example:

```
strcpy( "abc", buffer );
```

draws a diagnostic because **strcpy** is declared within **string.h** (by all the major compiler venders) as

```
char * strcpy( char *, const char * );
```

The diagnostic is issued because a **const char *** is being passed to a **char ***.

You may think it odd that string constants are not **const char *** by default. If you set this flag ON, you will probably discover the reason. There will undoubtedly be numerous places where a function is passed a string constant where the corresponding parameter should be declared **const char *** but isn't. There will also be cases of variables that should be declared as **const char *** but aren't. Thus, you may regard this flag as a good way to ferret out places where such type checking can be tightened.

fsg **std** is Global flag (default OFF).
Options: **+fsg** **-fsg** **++fsg** **--fsg**
If this flag is ON, names scoped with **std** will be searched for in **namespace std** primarily and the global scope secondarily. This is non-standard but some compilers exhibit this behavior. In particular, the GCC compiler does and so the option specifying this compiler (**-cgnu**) sets this flag ON automatically.

fsh SHared reading flag (default ON).
Options: **+fsh** **-fsh** **++fsh** **--fsh**
When flag **fsh** is ON, files opened for reading will be opened in shared mode. Currently this flag is implemented only on PC-lint.

fsn Strings as Names flag (default ON).
Options: **+fsn** **-fsn** **++fsn** **--fsn**
When the flag is OFF, the **esym()** option can be used only to suppress messages parameterized by the identifiers '***Symbol***' or '***Name***'. With this flag ON, one can use esym() to also suppress (or enable) messages parameterized by '***Context***', '***Kind***' , '***String***', '***Type***', and '***TypeDiff***'. For example:

```
//lint +fsn
//lint -esym(648,unsigned shift left)

int f()
    {
    return 20000u << 8; // Potential 648 ("Overflow in computing
                        // constant for operation 'String'")
    }
```

The **-esym()** second argument means that the **648** will not be issued when the operation (represented by the '***String***' parameter) is "**unsigned shift left**". Note that we can also disable this message for all bit shifts with:

```
//lint -esym(648,*shift*)
```

fsp SPecific flag (default ON).
Options: **+fsp -fsp ++fsp --fsp**
If this flag is ON (it is by default), Specific function call walking is supported. See Section 10.2.2 Interfunction Value Tracking. By turning this flag OFF (using the option **-fsp**), processing can be speeded up.

fss Sub Struct flag (default ON).
Options: **+fss -fss ++fss --fss**
This flag is intended to support a base-derived relationship among C structures similar to the same notion in C++. It is intended mostly for C programs but can be used by C++ programs that manipulate plain old data (POD) structures. If this flag is ON in a C program or has the value 2 in a C++ program (using the pair of options **+fcp ++fcp**) then the following definitions apply. A struct is considered a substructure of another if it is the type of the first member of that other struct. If a struct is a substructure of a second then it is treated for type comparison purposes as a base class of the second. In particular you will not receive Info **740** (Unusual pointer cast) when assigning pointers to struct's that are related as substructures. For example:

```
struct A { int a; };
struct B { struct A a; int b; };
void f( struct A *pa, struct B *pb )
   {
   pa = (struct A* ) pb;     /* no 740 */
   pb = (struct B* ) pa;     /* Info 826 but no 740 */
   }
```

fsu String Unsigned flag (default OFF).
Options: **+fsu -fsu ++fsu --fsu**
This flag has the same effect as **fcu**. The original intent was to specify that, with this flag ON, a string of constant characters (as in "...") would be regarded as a pointer to an unsigned character. It was subsequently decided that this shouldn't be specified independently of "..." expressions or of the signedness of the **char** type. It is retained for historical purposes.

fsv track Static Variable flag (default ON).
Options: **+fsv -fsv ++fsv --fsv**
If the flag is ON, static variables will be tracked. Since the flag is ON by default, this flag can be used to turn off static variable tracking (using **-fsv**).

Tracking static variables begins with pass 2, so to obtain static variable checking it is necessary to have at least two passes, e.g. **-passes(2)**. Information gleaned in pass 1 is used in later passes to determine the effect, if any, a given call is likely to have on given variables.

The external variables tracked by any function will depend on what variables a function uses and what variables its called functions use to within a certain depth. See option **-static_depth(*n*)**.

ftf raw Template Function flag (default OFF).
Options: **+ftf -ftf ++ftf --ftf**
A raw template function is a template function in the absence of specific argument types. If this flag is ON, raw template functions are parsed. This can assist a programmer who is in the process of developing a template by providing an "early warning system" before the template is actually instantiated. However, it can occasionally lead to syntactic difficulties such as when a parameterized type refers to a class whose name space does not yet exist. If the flag is OFF, this parsing is inhibited. In any event (i.e. for either flag setting) function instantiations are always parsed for each type instance. Note, a C++ compiler will not normally process raw template functions.

ftg TriGraph flag (default ON).
Options: **+ftg -ftg ++ftg --ftg**
If this flag in ON (it is ON by default) standard C/C++ trigraphs are permitted. For example **??(** is a trigraph that denotes the left square bracket (**[**). If this flag is OFF, use of the trigraphs will result in a diagnostic.

ftr TRuncate flag (default OFF).
Options: **+ftr -ftr ++ftr --ftr**
The effect of this flag is to truncate long include file names to 8x3 if the original filename does not seem to be valid. For example, consider the following:

```
#include <algorithms.h>
```

If **algorithms.h** cannot be opened using the usual search procedures, and if the Truncate flag is set (**+ftr**), then there will be an attempt to open the file '**algorith.h**'. This flag was introduced to allow for the peculiarities of Borland's C++ Builder.

ful **unsigned long** flag (default ON).
Options: **+ful -ful ++ful --ful**
If the unsigned long flag is ON, then **unsigned long** is a valid type. In K&R C **unsigned long** was not a valid type.

fus **using namespace std** flag (default OFF).
Options: **+fus -fus ++fus --fus**
If this flag is ON, the names within **namespace std** will be recognized without the need for a **using namespace std;** statement. This is non-standard but some compilers exhibit this behavior. In particular, the GCC compiler does and so the option specifying this compiler (**-cgnu**) sets this flag ON automatically.

fva[*N*] Variable Arguments flag (default OFF).

Options: **+fva[*N*]** **-fva[*N*]**

This option only affects functions that are NOT prototyped. Functions declared or defined while this flag is ON are assumed to have a variable argument list. Warning messages (**515** and **516**) reporting inconsistencies between argument lists are suppressed for such functions.

For example,

```
/*lint +fva */
extern int printf();
extern int fprintf();
/*lint -fva */
```

will cause **printf()** and **fprintf()** to be regarded as having variable argument lists.

An integer suffix ***N*** can be added to '**fva**' to denote that variability begins after the ***N***th argument. For example:

```
//lint +fva1
extern printf();
//lint -fva
```

indicates that only the first argument of **printf()** should be checked. Note that the same effect can be achieved by using prototypes.

A function, once dubbed as having variable argument status, cannot lose this status by being declared or defined with the **fva** flag OFF. This allows setting the flag once in one declarations module and omitting this flag in subsequent modules.

Note that the flag has no direct effect when a function call is encountered. That is, a function called with the flag ON will not be marked as having variable argument status. Whether an error is reported will depend on whether the function had been defined or declared with the flag having been ON.

fvl Variable Length array flag (default OFF).

Options: **+fvl** **-fvl** **++fvl** **--fvl**

If this flag is ON, we support variable length arrays (defined in **C99**) even under circumstances where we would not normally support them such as when processing C++ code. This flag is automatically turned on when we encounter the **-cgnu** option as GCC supports this feature for C++. If you want us to complain about the use of variable length arrays within C++ code while you are using the GCC compiler, you should follow the **-cgnu** option with a **-fvl** flag.

fvo Void data type flag (default ON).

Options: **+fvo -fvo ++fvo --fvo**
If this flag is ON, **void** is recognized as a type and functions declared as **void** are assumed to return no value. The original K&R C did not have a **void** type.

fvr Varying Return mode flag (default OFF).
Options: **+fvr -fvr ++fvr --fvr**
The return mode has to do with whether particular functions do, or do not, return a value. If this flag is ON when a function is defined or declared, then the function does not have to be consistent in this respect. Error messages arising out of an incompatibility between calls to the function and the function declaration or between two calls or between return statements and either of the above are inhibited. For example, since **strcpy()** returns a string (in most standard libraries) and since the string is seldom used, it would be wise to set this flag ON for at least one of the declarations of **strcpy()**.

This flag, once widely used, is now being replaced by the more concise:

-esym(534, ***Symbol1, nSymbol2,*** **...)**

fwc **wchar_t** is built-in flag (default ON).
Options: **+fwc -fwc ++fwc --fwc**
The **wchar_t** flag is ON by default. This flag has three states. If ON, **wchar_t** is built-in and becomes a reserved word for both C and C++. If OFF, say, by using the option:

-fwc

wchar_t becomes a built-in type for C++ modules only. This will be true even for a mixed suite of C and C++ modules. If it is set to a negative value, such as with the following option sequence:

-fwc --fwc

then **wchar_t** is not built-in for either dialect. Moreover, if **wchar_t** is built-in and a **typedef** for **wchar_t** is encountered then no disruptive message is issued and the **typedef** is ignored.

These rules represent a slight change to the rules for using this flag in earlier versions of our product. The old rule had only two states: ON meant the same as it does now and OFF meant no built-in **wchar_t** in either dialect. There was no way to automatically distinguish between C and C++.

The reason for the change can be summarized as follows. The two standards (C and C++) differ with respect to the reserved word **wchar_t**. In C++ it is built-in and in C it is obtained via a **typedef**. However, from a mixed mode (C and C++) point of view the

standards approach is not ideal since a variable declared **wchar_t** in C will have a different type than the same variable declared in the same way in C++. This difference would have to be reported.

Since a **typedef** of **wchar_t** is ignored when it is built-in we can predefine the built-in value for both C and C++. No diagnostic will be issued when a **typedef** for **wchar_t** is encountered in the C case. The only situation that might be affected adversely is the C-only programmer who might notice that the size and/or signedness of **wchar_t** has changed. This can be dealt with either by using the **-fwc** option (allowing a **typedef** to govern the exact type of **wchar_t**) or by adjusting the size and/or signedness of the built-in **wchar_t** by suitable settings of **-sw#** to indicate the size and **+fwu** to indicate signedness.

fwm **wprintf** formatting follows Microsoft flag (default OFF).
Options: **+fwm** **-fwm** **++fwm** **--fwm**
This flag is normally OFF. It is turned ON within compiler options files for the Microsoft compiler.

Format processing under Microsoft using the **wprintf** and **scanf** families of functions are slightly inconsistent with the standard. For example

```
wprintf( L"%s", "abc" );        // #1
```

is consistent with the standard but is inconsistent with the library for the Microsoft compiler. For Microsoft one may use one of :

```
wprintf( L"%S", "abc" );        // #2
wprintf( L"%hs", "abc" );       // #3
```

Neither of these is standard. With this flag ON we will regard #1 as erroneous and will not complain about #2 and #3.

If you want something that will work under both the standard and Microsoft you may try the following. Since the '**%S**' format is unique to Microsoft, the best strategy is to use #3. Here you would be relying on the library accepting the '**h**' without complaint. To print a wide character string you may want to use:

```
wprintf( L"%ls", L"abc" );      // #4
```

The length qualifier '**l**' is not needed with Microsoft but it will be accepted by Microsoft and the standard as well as imply a wide-character string.

Similar remarks can be made about the printing of characters with **printf** using the '**%c**' formatting code. The two forms:

```
wprintf( L"%hc", 'x' );         // #5
```

```
wprintf( L"%lc", L'x' );      // #6
```

seem to be happy compromises. The use of '**%C**' (i.e. the upper version) should be shunned as it is Microsoft only and operates counter-intuitively for **wprintf**.

A similar inconsistency is present with regard to **wscanf** and the same techniques should work with that family of function.

fwu **wchar_t** is Unsigned flag (default OFF).
Options: **+fwu** **-fwu** **++fwu** **--fwu**
This flag is used to specify the signedness of a built-in **wchar_t** type. If this flag is set ON the built-in type is of the unsigned variety, otherwise of the signed variety. A side-effect of this flag is to set the **+fwc** flag ON. Caution: Enabling **wchar_t** as a reserved word will render invalid any **typedef** of that name.

fxa eXact Array flag (default OFF).
Options: **+fxa** **-fxa** **++fxa** **--fxa**
(This option deals only with non-prototype functions). This flag, if ON, selectively inhibits promotion of array arguments and array parameters (for the purpose of type matching) to pointers. This provides a more strict type-checking in function calls than is normally obtainable. In particular, only arrays may be passed to parameters declared as array and the sizes, if specified, must match. On the other hand, both arrays and pointers may be passed to a parameter typed as pointer. See Section 13.7 Exact Parameter Matching

fxc eXact **char** flag (default OFF).
Options: **+fxc** **-fxc** **++fxc** **--fxc**
(This option deals only with non-prototype functions). This flag, if ON, inhibits promotion of **char** or **unsigned char** arguments and parameters (for the purpose of type matching). Normally these types are silently promoted for argument passing to **int**, and this promotion can hide unintended disagreements between parameter and argument. See Section 13.7 Exact Parameter Matching

fxf eXact Float flag (default OFF).
Options: **+fxf** **-fxf** **++fxf** **--fxf**
(This option deals only with non-prototype functions). This flag, if ON, inhibits promotion of **float** arguments and parameters (for the purpose of type matching). Normally these types are silently promoted to **double**, and this promotion can hide unintended disagreements between parameter and argument. See Section 13.7 Exact Parameter Matching

fxs eXact Short flag (default OFF).
Options: **+fxs** **-fxs** **++fxs** **--fxs**
(This option deals only with non-prototype functions). This flag, if ON, inhibits promotion of **short** and **unsigned short** arguments and parameters (for the purpose of type matching). Normally these types are silently promoted to **int**, and this promotion

can hide unintended disagreements between parameter and argument. See Section 13.7 Exact Parameter Matching

fzl siZeof is Long flag (default OFF).
Options: **+fzl** **-fzl** **++fzl** **--fzl**
If this flag is ON, **sizeof()** is assumed to be a **long** (or **unsigned long** if **-fzu** is also ON). The flag is OFF by default because **sizeof** is normally typed **int**. This flag is automatically adjusted upon encountering a **size_t** type. This flag is useful on architectures where **int** is not the same size as **long**.

If the flag has a value equal to 2, then **sizeof()** is assumed to be **long long**. Thus

```
+fzl    ++fzl
```

will result in **sizeof()** being **unsigned long long x** (assuming **+fzu**).

fzu siZeof is Unsigned flag (default ON).
Options: **+fzu** **-fzu** **++fzu** **--fzu**
If this flag is ON, **sizeof()** is assumed to return an unsigned quantity (**unsigned long** if **-fzl** is also ON). This flag is automatically adjusted upon encountering a **size_t** type.

5.6 Message Presentation Options

PC-lint/FlexeLint allows considerable control over the presentation of messages.

5.6.1 Message Height Option

PC-lint/FlexeLint allows considerable control over the height of error messages. The smaller the height, the more messages can be squeezed into a smallish screen image; the greater the height, the greater the clarity of error presentation.

Message height is controlled by the **-h** option having the general form:

-h[s/S][F][f][a][b][r][mn/e][m][m/*M*/][I]*N*

The optional **s** means Space (blank line) after each message. However, **-hs** does not skip a line after wrap-up messages. The **-hS** (note the CAPITAL '**S**') option will force a line skip in both places.

The optional **F** means that an error message is ***always*** provided with File information. This is useful when error messages are processed automatically by, say, an editor. If the error condition is detected after the last line of a module, the message is tagged with an appropriate line in some file. If no such line seems appropriate, it will be tagged with line ***n***+1, where ***n*** is the number of

lines in the module. This is useful for some editors that automatically place themselves at the appropriate place in the file.

The optional **f** is similar to **F**. It will add out-of-sequence file position information only if it seems especially useful but it will not resort to the ***n***+1'st line of a file. Use '**F**' if your editor can accept an ***n***+1'st line; if not use '**f**'.

The **a** and **b** (meaning respectively Above and Below) refer to the location of the indicator ***I*** with respect to the source line. This is used only for heights of 3 and 4.

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

The optional **mn** requests No Macro display. Normally, the value of each open macro at the time of the error is shown on a separate line, provided the message height (i.e. the value of ***N***) is at least 3. The line is by default prefixed with "**#...**". For example, the following code

```
#define foo(a,b) ((a)/(b)+3)
int n = foo(4,0);
```

can result in the following message:

```
#...   ((4)/(0)+3
int n = foo(4,0);
file.c  3 Error 54: Division by 0
```

The line beginning with "**#...**" is part of the macro display. The macro display can be multi-line and for long macros will be truncated fore and aft. The macro display is to aid diagnosis and is suppressed with **mn**. The optional **m** will undo the effects of **mn**; i.e. it will restore the display of macros. Meanwhile, the optional **e** places the source line (and the indicator and the macro expansion, if any) at the end of the message. The optional **m/*M*/** restores the macro display and lets you assign a new prefix ***M***, in place of the default "**#...**". A convenient way to place a space within the string ***M*** is via **\s**. To place a '**/**' within ***M*** use **\/**.

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

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

The default height option is **-ha_3**

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

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

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

Example (**-hb^4**):

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

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

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

Example (**-hb^3**):

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

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

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

Example (**-h$2**):

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

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

Example (**-h1**):

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

5.6.2 Message Width Option

The format of the width option is:

-width(*W*,*Indent*)

Example: **-width(99,4)**

The first parameter (***W***) specifies the width of messages. Lines greater than this width are broken at blanks. A width of **0** implies no breaking. The second number specifies the indentation to be used on continued lines. **-width(79,4)** is the default setting.

5.6.3 Message Format Options

-format=...

controls the detailed format of messages where the message height is three or less.

-format4a=...

-format4b=...

controls the detailed format of messages where the message height is four. See below for details.

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

%f = the filename
(note that option **+ffn**, standing for "Full File Names",
can be used to control whether full path names are used).
%l = the line number
%t = the message type (Error, Warning, etc.)
%n = the message number
%m = the message text
%c = the column number
%C = the column number +1
%i = the invoking function

%% = a percent sign
%(...**%)** = conditionally include the information denoted
by ... if the error occurred within a file.
\n = newline
\t = tab
\s = space
\a = alarm (becomes ASCII 7)
\q = quote (**""**)
**** = backslash ('\')

For example the default message format is

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

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

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

-format_specific The prologue to a Specific Walk error message is controlled by this option. See Section 10.2.2.4 Specific Walk Options Summary

-format_stack=... controls the detailed formatting of the output produced by **+stack**. This is the report that deals with stack usage. If this option is not given, a default is assumed.

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

The format string may contain the following escapes:

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

\n = newline
\t = tab
\s = space

\a = alarm
\q = quote(")
**** = backslash

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

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

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

-format_summary=... controls the detailed formatting of the output produced by the **-summary** option. See **-summary** for details.

-format_template=... controls the detailed formatting of a prologue to any message issued while instantiating a class template. The default value for this option is:

```
-"format_template=\n%{  While instantiating %i at File %f line %l\n%}"
```

The following additional escape sequences are applicable:

%{...%} designates a repetition, one for each class template currently in the process of being instantiated.

%i identifies the template class being instantiated.

For example the following code

```
template< class X > class A
     { static const X x = 3; };
template< class T > struct C { A<T> aa; };
C<double*> z;
```

can result in the following message.

```
While instantiating struct C<double *> at File a4.cpp line 18
While instantiating A<double *> at File a4.cpp line 14
                                    _
       static const X x = 3;
a4.cpp(6) : Error 64: Type mismatch (initialization) (double * = int)
a4.cpp(14) : Info 831: Reference cited in prior message
a4.cpp(18) : Info 831: Reference cited in prior message
```

The first two lines are controlled by the information between the **%{...%}** brackets. The Informational **831** messages are inserted as a result of encountering the **%f %l** escapes.

-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. An example of its use can be found in the file **env-html.lnt**.

The format string may contain the following escapes:

%m = the normal verbosity message

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

5.6.4 Appending Text to Messages

-append(*errno*,*string*)

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

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

will append the indicated message to the text of message **936**.

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

The string appended to a message may include the backslash escapes available with the **-format** option. For example, the option:

```
-append( 936, \n\tSee Corporate Software Policy 371 )
```

will place the appended text indented on a separate line. Other escapes are **\q** (quote), **\s** (space) and **\a** (alarm). See **-format** in section 5.6.3 Message Format Options.

Note that this option does not automatically enable the indicated message. This would be done separately with, in this example, the option **+e936**.

Additionally, the option can be parameterized to append the given text only when certain names appear in the Lint output. For example:

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

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

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

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

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

5.7 Other Options

-A requests strictly ANSI/ISO C/C++ processing. Non-standard keywords (i.e., reserved words) and other non-standard features are reported upon but duly processed according to their non-standard meaning.

-A(*LanguageYear*)

You can specify the version of the language you are using with this **-A** option. Examples:

```
-A(C90)          // specifies C 90
-A(C++2003)      // specifies C++ 2003
```

The only languages permitted to be specified are C and C++. This is followed either by a two digit year (with '19' or '20' prepended as appropriate) or a four digit year. It is not necessary to specify the precise year of a standard. For C, any year that precedes 1999 is assumed to be specifying the C90 standard. For C++ any year preceding 2003 is assumed to be specifying the 1997 standard. There will be a new standard for C++ sometime around 2010. By default we will always assume the latest standard.

You may, of course, specify the versions of both C and C++. The language is deduced from the file name extension and not from this 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. There are a number of options:

-align_max(*n*) where ***n*** is an integer, sets the maximum alignment to be ***n***. The alignment of any object will be the maximum of ***n*** and the alignment of its type (as established by the **-a**... options). If ***n*** is 1, data will be aligned on a byte boundary.

I.e. it is said to be packed. If ***n*** is 0, the temporary maximum alignment is ignored. The default value of the maximum alignment is 0. If the argument to the option is missing, 0 is assumed.

-align_max(push) will push the current maximum alignment onto a stack. The current maximum is not changed.

-align_max(pop) will restore the current maximum alignment to the value currently at the top of the alignment stack and pop the stack.

For example, assume that the alignment of integers is 4 (**-ai4**), the size of integers is 4 (**-si4**) and the size of characters is 1. Then the following code:

```
//lint -align_max(push) -align_max(1)
struct A { char a; int b; };
//lint -align_max(pop)
struct B { char a; int b; };
```

will result in **struct A** having a size of 5 and in **struct B** having a size of 8.

-atomic(*type-pattern*) Any type matching the type-patterns is considered an atomic type. Reads and writes of atomic types are considered to be atomic operations. See **Section 12.9.2 Atomic Types**.

-b Suppresses the Banner line
+b Redirects the Banner line (to standard out)
++b Places the banner line onto **-os(*file*)**
(Unlike most other options, this option must be placed on the command line and not in an indirect file.) When PC-lint/FlexeLint is run from some environments, the banner line (identifying the version of PC-lint/FlexeLint and bearing a Copyright Notice) may overwrite a portion of an editing screen. This is because the banner line is, by default, written to standard error whereas the messages are written to standard out and can be redirected. The option **+b** will cause the banner line to be written to standard out (and hence will become part of the redirected output). The option **-b** will suppress the banner line completely.

The option **+b** works well for:

```
lint +b ... >outfile
```

Unfortunately this will not have the intended effect with:

```
lint +b -os(outfile) ...
```

as the banner line is written before the **-os** option has had a chance to take effect.

++b will deposit the banner line into standard out anywhere it is encountered. Thus:

```
lint -os(outfile) ++b ...
```

will cause the banner line to be placed into **outfile**. You will also get a banner line in standard error but this can be separately suppressed as in:

```
lint -b -os(outfile) ++b ...
```

-background This option, under Windows, will have the effect of running PC-lint in the background. That is, it will run at a reduced priority.

-c*code* specifies a particular compiler. For a list of codes and the effect that each of these codes has, see Section 5.8.2 Compiler Codes. This option is required for MS-DOS because the set of pre-defined preprocessor identifiers depends on the memory model selected. Options for other systems can be governed by a compiler options file. See Section 5.8 Compiler Adaptation

+cpp(*ext*) Add C++ extension
-cpp(*ext*) Remove C++ extension
This option allows the user to add and/or remove extensions from the list that identifies C++ modules. By default only **.cpp** and **.cxx** are recognized as C++ extensions. (It is as if **+cpp(cpp,cxx)** had been issued at the start of processing.) For example:

```
lint a.cpp +cpp(cc) b.cc c.c
```

treats **a.cpp** and **b.cc** as C++ modules and **c.c** as a C module. There is no intrinsic limit to the number of different extensions that can be used to designate C++ modules. See also flag **+fcp**.

Note: If you are using **+cpp(.C),** i.e. you want to use case to distinguish C++ vs. C on Windows, you need to also turn off the fold file name flag (**-fff).**

-d... This option allows the user to define preprocessor variables (and even function-like macros) from the command line. The simplified format of this option is:

-d*name*[=*value*]

where the square brackets imply that the value portion is optional. If **=*value*** is omitted, 1 is assumed. If only ***value*** is omitted as in **-dX=** then the ***value*** assigned is null. For function-like macros, see option **-d*name*()=*Replacement*** in Section 5.8.3 Customization Facilities. Examples:

```
-dDOS
-dalpha=0
-dX=
```

These three options are equivalent to the statements

```
#define DOS 1
#define alpha 0
#define X
```

appearing at the beginning of each subsequent module.

Note that case is preserved. There is no intrinsic limit to the number of **-d** options. The **-d** option may be used within a **/*lint** comment; it will take effect in the current module and all subsequent modules. See also the **-u**... option and Section 5.1 Rules for Specifying Options.

This option does not provide any functionality over what can be provided through the use of **#define** within the code. It does allow lint to be customized for particular compilers without modifying source. It also applies globally across all modules, whereas **#define** is local to a specific module.

See also **-u** and **--u**.

+d*name*=[***value***] An option of the form **+d**... behaves like **-d**... except that the definition is locked in and will be resistant to change even though a subsequent **#define** of the same name is encountered. For added security **++d*name*=*value*** will behave in a similar fashion and, moreover, name cannot be **undef**'ed.

Using this option you can lock in the definition of function-like macros as well as object macros.

For example, suppose the PC-lint/FlexeLint is stumbling badly over the macro

```
offsetof(s,m)
```

First place your definition within a header file under a slightly different name:

```
#define my_offsetof(s,m) some definition...
```

Then use the options:

```
+doffsetof=my_offsetof
-header( my_offsetof.h )
```

where **my_offsetof.h** contains the definition of the **my_offsetof** macro.

You may also explicitly set function-like macros. See **-d*name*(**[***list***]**)** in **Section 5.8 Compiler Adaptation.**

See also **-u**... and **--u**...

-D*name*[**=***value*][**;** ...]
The **-D** option is similar to the **-d** option except that a semi-colon separated list of ***name-value*** pairs is supported. Thus:

```
-Dalpha=2;beta;gamma=delta
```

is equivalent to:

```
-dalpha=2  -dbeta  -dgamma=delta
```

The reason for this option is to allow cooperating interactive development environments (IDE's) to pass on to language processors (like lint) the define directives selected by the user that are otherwise intended for the compiler. For example, under the Borland C++ 5.00 IDE, you can use the option

```
-D$DEF
```

to pass a collection of define directives on to PC-lint.

+D*name*[**=***value*][**;** ...]
The **+D** option is similar to the **+d** option except that a semi-colon separated list of ***name-value*** pairs is supported. Thus:

```
+Dalpha=2;beta;gamma=delta
```

is equivalent to:

```
+dalpha=2  +dbeta  +dgamma=delta
```

The reason for this option is to allow cooperating interactive development environments (IDE's) to pass on to language processors (like lint) the define directives selected by the user that are otherwise intended for the compiler. For example, under the Borland C++ 5.00 IDE, you can use the option

```
+D$DEF
```

to pass a collection of define directives on to PC-lint.

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

category is one of:

function
keyword
macro
variable

The ***commentary*** in the third argument will be appended to the message. For example,

```
-deprecate( variable, errno, Violates Policy XX-123 )
```

When the use of **errno** as a variable is detected (but not its definition or declaration) the following Warning is issued.

```
Warning 586: variable 'errno' is deprecated. Violates Policy XX-123
```

When the category of deprecation is **variable** only the use of external variables are flagged. Local variables may be employed without disparaging comment.

If **errno** were a **macro** you would need to deprecate **errno** as a **macro**:

```
-deprecate( macro, errno, Violates Policy XX-123 )
```

If **errno** could be either (the standard allows both forms) then both options should be used.

You may also deprecate functions and keywords (as the list above suggests). For example:

```
-deprecate( keyword, goto, goto is considered harmful )
-deprecate( function, strcpy, has been known to cause overuns )
```

could be used to flag the use of suspect features.

A limiting factor on the length of commentary is the maximum size of any one option, which at this writing is 600 characters. Quotes (both single and double) and parentheses within the commentary need to be balanced.

+ext(***extension*** [, ***extension***] ... **)**
specifies the list of default extensions tried by PC-lint/FlexeLint and the order in which they are tried when an extension-less filename is provided as argument. For example,

```
lint alpha
```

will, by default, cause first an attempt to open **alpha.lnt**. If this fails there will be an attempt to open **alpha.cpp**. If this fails there will be an attempt to open **alpha.c**. It is as if the option:

```
+ext( lnt, cpp, c )
```

had been given on startup.

Minor notes: This has no effect on which extensions indicate that a module is to be regarded as a C++ module. This is done by the options **-/+cpp** and **-/+fcp**. Prefixing an extension with a period has no effect. Thus, **+ext(lnt,c)** means the same as **+ext(.lnt,.c)** . For MS-DOS and Windows, upper-casing the extension also has no effect. Thus, **+ext(lnt)** has the same effect as **+ext(LNT)**. On Unix, however, case differences do matter. For example, if the Unix programmer wanted both **.c** and **.C** extensions to be taken by default he might want to use the option: **+ext(lnt,c,C)**

-fallthrough

indicates that the programmer is aware of the fact that flow of control is falling through from one case (or default) of a switch to another. Without such an option Message **825** will be issued. For example:

```
case 1:
case 2:  n = 0;  // setting n to 0
case 3:  n++;
```

will result in Info **825** on case 3 because control is falling through from the statement above, which is neither a case nor a default. The cure is to use the **-fallthrough** option:

```
case 1:
case 2:  n = 0;  // setting n to 0
//lint -fallthrough
case 3:  n++;
```

Warning **616** will be issued if no comment at all appears between the two cases. If this is adequate protection, then just inhibit message **825**.

-father(***Parent***, ***Child*** [, ***Child***] ... **)**

is like the **-parent()** option except that it makes the relationship a strict one such that a ***Child*** type can be assigned to a ***Parent*** type but not conversely. To make all relationships strict you may use the **-fhd** option. (Turn off the Hierarchy Down flag). See Section 5.5 Flag Options If a **-parent()** option and a **-father()** option are both given between the same two types then the relationship is considered strict. See Section 9.6.4 Restricting Down Assignments (-father)

-function(***Function0***, ***Function1*** [, ***Function2***] ... **)**

This option specifies that ***Function1***, ***Function2***... are like ***Function0*** in that they exhibit special properties normally associated with ***Function0***. The special functions with built-in meaning are described in Section 11.1 Function Mimicry (-function). See also **-sem** in Section 11.2 Semantic Specifications.

-header(*filename*) The **-header** option will force PC-lint/FlexeLint to read the header ***filename*** at the outset of each module. This is useful for defining macros, **typedefs**, etc. of a global nature used by all modules processed by PC-lint/FlexeLint without disturbing any source code. For example,

```
-header( lintdcls.h )
```

will cause the file **lintdcls.h** to be processed before each module.

The header is not reported as being unused in any given module (even though it may be). It is not considered a library header. An extra option may be needed to make this assertion as follows: **+libh(*filename*)**

Multiple **-header** options may be used, and this effect is additive. Files are included in the order in which they are given. However, an option of the form:

```
--header( filename )
```

will remove all prior headers specified by **-header** options before adding ***filename***.

If ***filename*** is absent as in **--header** then the effect is to erase any prior header requests.

+headerwarn(*filename* [, *filename*]…)
will cause a message (Info **829**) to be issued each time one of the ***filename*** arguments is included as a header. For example **+headerwarn(stdio.h)** will alert the programmer to the use of **stdio.h**. If the option **–wlib(1)** is in place, as it usually is to stem the flood of Warnings and Informationals emanating from library headers, no message **829** will be issued from within a library header unless you also issue a **+elib(829)** sometime after the **–wlib(1).**

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

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

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

is the normal **stderr** or, if the **-oe(*filename*)** option is given, the destination designated by that option.

The sub-options are:

version(*html-version*) can be used to designate the version of HTML. Its use is optional. The version identification will be placed within angle brackets and output before the **<html>** at the start of the output file.

head(*file*) is another option argument and can be used to supply header information for the HTML output. The file is searched for (in the usual places as if it had been specified on a **#include** line) and copied into standard output just after the line that contains "**<html>**" that normally begins an HTML file.

-i*directory* Files not found in the current directory are searched for in the directory specified. There is no intrinsic limit to the number of such directories. The search order is given by the order of appearance of the **-i*directory*** options. For example:

```
-i/lib/
```

can be used (on Unix-like operating systems) to make sure that all files not found in the current directory are looked up in some library directory named **lib**. A file separation character ('\' for MS-DOS, '/' for Unix, ':' for VAX VMS) will be appended onto the end of the **-i** option if a file meta character ('\' or ':' or '/' for MS-DOS, '/' for Unix, ':' or ']' for VAX VMS) is not already present. Thus

```
-i/lib
```

is equivalent to the above (for Unix). See FlexeLint Installation Notes for any system specific details.

To include blanks within the directory name employ the quote convention (See Section 5.1 Rules for Specifying Options) as in the following:

```
-i"program files\compiler"
```

Multiple directories may be specified either with multiple **-i** options or by specifying a semi-colon separated list with a single **-i** option. (See also **+fim**)

PC-lint/FlexeLint also supports the **INCLUDE** environment variable. See Section 15.2.1 INCLUDE Environment Variable Note: Any directory specified by a **-i** directive takes precedence over the directories specified via the **INCLUDE** environment variable.

For Unix users, **-I*directory*** is identical to **-i*directory***.

As a special case the option **-i-** is taken as a directive to remove all of the directories established with previous **-i** options (it has no effect on those directories specified with **INCLUDE**).

--i*directory* This is like **-i*directory*** but places a lower priority on that directory. All directories specified by **-i** are searched before directories named by **--i**. This is to support compilers that always search through compiler-provided library header directories after searching user-provided directories.

Example: suppose there is a header file named '**bar.h**' in both directory '**/foo**' and directory '**local**'. Then:

```
// in std.lnt:
--i/foo          // search foo with low priority
-ilocal          // search local with high priority
// in t.cpp:
#include <bar.h> // finds the version in 'local'
```

-I- for Sun CC. After Lint processes the option **-csun**, it will behave as Sun CC does when it encounters the **-I-** option (Refer to the Sun C++ User's Guide for details). After this option is given, quote style headers will not be searched for in the directory of the including file, and angle bracket header files will be searched for only in directories that are mentioned in **-i** options after the **-I-**.

-ident(*string*)
This option allows the user to specify alternate identifier characters. Each character in ***string*** is taken to be an identifier character. For example if your compiler allows **@** as an identifier character then you may want to use the option:

```
-ident(@)
```

Option **-$** is identical in effect to **-ident($)** and is retained for historical reasons.

-ident1(*string*)
This option allows the user to define a one-character identifier. A one-character identifier is one that is not part of some other identifier. E.g.

```
-ident1(@)
```

will establish "**@**" as being a one-character identifier. Thus, **@abc** will consist of two identifiers "**@**" and "**abc**". A one-character identifier is a unique lexical unit. It differs from a regular identifier character, such as a letter, in that it does not join with other characters and it differs from a special character (like comma) in that it can be used as the name of a macro.

-idlen(*count* [, *options*])

will report on pairs of identifiers in the same name space that are identical in their first ***count*** characters but are otherwise different. ***Options*** are:

- **x** linker (eXternal) symbols
- **p** Preprocessor symbols
- **c** Compiler symbols

If omitted, all symbols are assumed.

Frequently, linkers and, less frequently, preprocessors and compilers, have a limit on the number of significant characters of an identifier. They will ignore all but the first ***count*** characters. The **-idlen** option can be used to find pairs of identifiers that are identical in the first ***count*** characters but are nonetheless different. PC-lint/FlexeLint treats the identifiers as different but reports on the clash.

Option **x**, linker symbols, refers to inter-module symbols. Option **p**, preprocessor symbols, refers to macros and parameters of function-like macros. Option **c**, compiler symbols, refers to all the other symbols and includes symbols local to a function, **struct**/**union** tags and member names, **enum** constants, etc. Warning **621**, **Identifier clash** may be suppressed for individual identifiers with the **-esym** option. **-idlen** is off by default.

-incvar(***name*** **)**

The environment variable **INCLUDE** is normally checked for a list of directories to be searched for header files (See Section 15.2 include Processing). You may use the **-incvar(*name*)** option to specify a variable to be used instead of **INCLUDE**. For example

```
-incvar(MYINCLUDE)
```

requests checking the environment variable **MYINCLUDE** rather than checking **INCLUDE**.

Limitation: This option may not be placed in an indirect file or source file. It may be placed on the command line or within the **LINT** environment variable. The **INCLUDE** environment variable is processed just before opening the first file.

-index(***flags, ixtype, sitype*** [, ***sitype***] ... **)**

This option is supplementary to and can be used in conjunction with the **-strong** option. It specifies that ***ixtype*** is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type ***sitype*** (or ***sitype'***s if more than one is provided). Both the ***ixtype*** and the ***sitype*** are assumed to be names of types subsequently defined by a **typedef** declaration. See Section 9.5 -index

-indirect(***options-file*** [,...] **)**

allows you to specify Lint option files to be processed when this option is encountered. This is useful if you want to use an options file within a Lint comment. For example, Lint

option files that are appropriate only for a particular configuration may be conditionally included. The code below may appear in some header that is included either in every module or at least the first module. Thus, the header in question can be injected with the **-header** option.

Notice also the unusual include guard. We need to guard against this section of code being reprocessed in subsequent modules so we can't use the usual **#define** to define **BEEN_HERE**. This would go away in the next module. Instead the **-d** is sticky enough to come back each and every module.

```
#ifndef BEEN_HERE
    #if defined(HAS_LIBRARY_A)
        //lint -indirect(lib-a.lnt)
    #elif defined(HAS_LIBRARY_B)
        //lint -indirect(lib-b.lnt)
    #else
        //lint -indirect(lib-default.lnt)
    #endif
        //lint -dBEEN_HERE
#endif
```

+libclass(***identifier*** [, ***identifier***] ... **)**
specifies the class of header files that are by default treated as library headers. Each ***identifier*** can be one of:

angle	(specified with angle brackets),
foreign	(comes from a foreign directory using **-i** or the **INCLUDE** environment variable),
ansi	(one of those specified by ANSI/ISO C), or
all	(meaning all header files).

For more information, see Section 6.1 Library Header Files.

+libdir(***directory*** [, ***directory***] ... **)** Add library directory
-libdir(***directory*** [, ***directory***] ... **)** Remove library directory
allows you to override **+libclass** for particular directories. ***directory*** may contain wild cards ('*****' and '**?**'). For more information, see Section 6.1 Library Header Files.

+libh(***file*** [, ***file***] ... **)** Add library header
-libh(***file*** [, ***file***] ... **)** Remove library header
allows you to override **+libclass** and **+/-libdir** for particular headers. ***file*** may contain wild card characters. For more information, see Section 6.1 Library Header Files.

+libm(***module-name*** [,...] **)**
allows you to specify modules as library files. The ***module-name*** may contain wild card characters. For more information, see 6.1 "Library Header Files"

-library This option turns ON the library flag for the next module, if given on the command line, or for the rest of the module if placed within a lint comment. For an example, see Section 6.2 Library Modules. At one time this option was equivalent to the **+flb** flag (See Section 5.5 Flag Options). However, there is now a difference. **-library** designates the file in which it is placed and all files that it may include as having the library property. Thus, if a lint option within a header file contains the **-library** flag then only that header (and the headers it includes) are affected. It does not affect the including file. With **+flb** the flag is left on until turned off. This option is equivalent to the Unix lint option **/* LINTLIBRARY */** (See Section 13.9 Unix Lint Options). Some Unix headers contain this option and in doing so cause the header in which it is found to be considered library.

-limit(*n* **)** This option imposes an upper limit on the number of messages that will be produced. By default there is no limit. (The maximum value for *n* is 64000.

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

+linebuf This option doubles the size of the line buffer, which is initially 600 bytes. The line buffer is used to hold the current line and must be as long as the longest line in whatever file you are reading. Macro expansion does not affect this value. If a line exceeding this value is detected, Error **91** is issued and you will be prompted to use this option. There is no intrinsic limit to the number of times the line buffer can be doubled. The option can be placed in a file (specifically in a **.lnt** file). When the line buffer is reallocated, the old buffer is not released, so that information in the old buffer is still processed even though the line buffer is changing under foot.

+lnt(***extension*** [, ***extension***] ... **)** adds
-lnt(***extension*** [, ***extension***] ... **)** removes
an extension (or extensions) from the list of filename extensions used to indicate indirect files (by default only **lnt** designates an indirect file). For example, if you want files ending in **.lin** to be interpreted as indirect files you use the option:

```
+lnt( lin )
```

After such an option, a filename such as **alpha.lin** will be interpreted as if it had been named **alpha.lnt**. That is, it will be interpreted as an extension of the command line rather than as a C/C++ program.

This will not affect the sequence of default extensions that are tried. Thus, when the name **alpha** is encountered, there will not first be a test to see if **alpha.lin** exists. This is governed by the **+ext** option.

If you want to remove the name **lnt** and replace it by **lin** you need to use the pair of options:

```
-lnt(lnt)  +lnt(lin)
```

-lobbase(*filename*)
will provide a base for lob (Lint Object Module) production. The option has no effect unless a lob is being produced (**-oo** is being given, see below). In this case the lob produced will be the binary difference between the normal lob output and the base. This can save considerable space when a number of modules all include the same voluminous header file information as is the case for much GUI (Graphic User Interface) programming. See Chapter 8. LINT OBJECT MODULES.

+macros increases the size of macro storage. The largest size of any individual macro is by default 4096 bytes. If this size is exceeded, the fatal error **303** is issued. The option: **+macros** can be used to double this size. Repeating the option will again double the size and the doubling is repeated for each repetition of the macro. In a similar fashion, the option **-macros** will halve the size.

The option must be given before processing the first module. If given too late, error **75** will be issued alerting you to the fact that the option had no effect.

The size of a macro is a little tricky to describe. For example,

```
#define  A alphabet
#define  B(x) A x
```

Then the size of **B(abc)** is 5 (one for the **A**, one for the blank and 3 for the argument). Thus the count is taken after macro argument substitution and before macro replacement.

Yes, there are programmers using macros that are larger than 4096.

-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. For example,

```
-maxfiles(10000)
```

raises this limit to 10,000 files. This option must be given before the first module is processed. The most straightforward usage is to place the option within a **.lnt** file.

-maxopen(*n*) When a nested **#include** sequence becomes heavily nested PC-lint/FlexeLint starts closing down early files in order to reopen new ones. When a closed file is resumed it is reopened and **fseek** is used to position the file. There is no way in general to determine the maximum number of open files. By default, PC-lint and FlexeLint presume they can open 12 files simultaneously. However, some operating systems/libraries do not allow 12 files open simultaneously. If, for example, your system will allow only 8 files open simultaneously you may use the **-maxopen(8)** option to express this fact. As long

as the number given in the option is less than the actual number of allowed open files, you will be OK.

If PC-lint/FlexeLint cannot open as many files as it thinks it should be able to, then, with deeply nested include sequences, a Fatal Error **322** will be issued.

-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. The exact behavior of this option depends on whether or not it is used in a Lint comment. If it is used in a Lint comment, then the argument passed to it should be just as if you were using **#pragma** message. For example:

```
#define MAC 100
/*lint -message("current value of MAC is " MAC) */
```

yields an **865** informational message whose text is:

```
current value of MAC is 100
```

Note that in order to keep macros from being expanded, their names must appear in a double-quoted sequence. Also note that special characters such as single quotes, commas, and escape sequences must also appear inside a double-quoted sequence.

When not processing source files, **-message()** will not expand macros defined with the **-d** option, but it will expand environment variables. For example, you might put this in your **std.lnt**:

```
// (Assume INCLUDE has been set to 'C:\compiler\include')
-message(INCLUDE is set to: %INCLUDE%)
// For this option, Lint will print:
// Info 865: INCLUDE is set to: C:\compiler\include
```

Note that it is not necessary to use double-quotes when using **-message()** within an options file. Environment variable names will not be expanded so long as they are not surrounded with '%' characters.

When processing source files, **-message()** will not expand environment variables.

-mD specifies the **D** memory model (large Data, small program model). Pointers to data areas are assumed to be **far**. See Section 14.1 Memory Models

-mL specifies the Large memory model (pointers are assumed to be **far**). See Section 14.1 Memory Models

-mP specifies the **P** model (large Program, small data model, sometimes referred to as the medium model). Pointers to program areas (pointers to functions) are assumed to be **far**. See Section 14.1 Memory Models

-mS specifies the Small model (pointers are assumed to be **near**). This is the default and is provided for completeness. See Section 14.1 Memory Models

-od[**s**][**i**][**f**][***width***](***filename***)
Output Declarations (including prototypes) to ***filename*** using the optional ***width*** to specify the maximum line width. If **i** is specified, functions with Internal linkage are included; if **s** is specified, Structure definitions are provided and, if **f** is specified, output is restricted to Functions. [**s**][**i**][**f**] may appear in any order. See Section 13.6 Prototype Generation If **+od** is specified rather than **-od**, output is ***appended*** to the file named ***filename***.

-oe(***filename*** **)**
redirects output intended for Standard Error (**stderr**) to the named file. This is primarily used to capture the help screen. For example:

```
lint -oe(temp) +si4 ?
```

dumps the help information to file **temp** ***after*** the size setting has been made. If the option is introduced with a '**+**' as in **+oe(temp)** output is ***appended*** to the named file.

-ol(***filename*** **)**
Output Library information to ***filename***. A digested form of the information within a library module is output. Using this in place of the original library module will speed up subsequent processing. For example:

```
lint sl.c -ol(sl.c)
```

can be used to capture the library declarations specified within the file **sl.c** in concentrated form.

Note that you need a **-library** option embedded within the file **sl.c** (or equivalently preceding the file on the command line) or else nothing comes out. This option is largely superseded by **-oo**. See Chapter 8. LINT OBJECT MODULES and Section 6.2 Library Modules.

-oo[**(** ***filename*** **)**]
Output Object module to ***filename***. This option causes binary information for all processed modules (usually just one) to be output to ***filename***. The extension for ***filename*** should be **.lob**. If ***filename*** is omitted, as in **-oo**, a name will be manufactured using the first name of the source file and an extension of **.lob**. See Section 8.3 Producing a LOB Related options are **+fol**, **+fod** and **-lobbase**.

-os(***filename*** **)**
causes Output directed to Standard out to be placed in the file ***filename***. This is like redirection and has the following advantages: (a) the option can be placed in a **.lnt** file or

anywhere that a lint option can be placed (b) not all systems support redirection and (c) redirection can have strange side effects (See Section 8.5 Make Files). If **+os** is used rather than **-os**, output is appended to the file. Make sure this option is placed before the file being linted. Thus

```
lint -os(file.out) fil.c
```

is correct. But

```
lint fil.c -os(file.out)
```

loses the intended output. The reason is that the redirection doesn't start until the option is encountered.

-p[**(** ***width*** **)**] run just the Preprocessor. If this flag is set, the entire character of PC-lint/ FlexeLint is changed from a diagnostic tool to a preprocessor. The output is directed to standard out, which may be redirected. Thus,

```
lint -os( file.p ) -p file.c
```

will produce in **file.p** the text of **file.c** after all # directives are carried out and removed. This may be used for debugging to determine exactly what transformations are being applied by PC-lint/FlexeLint.

The optional argument (***width***) denotes an upper bound on the width of the output lines. For example:

```
-p(100)
```

will limit the width of output lines to 100 characters. Splitting is done only at token boundaries. Very large tokens will not be split even if they exceed the nominal line limit. This is so the result can be passed back through lint or some other C/C++ source processor.

In order to track down some complicated cases involving many include headers you may want to use the **-v1** verbosity option in connection with **-p**. Recall (Section 5.4 Verbosity Options) that **-v1** will produce a line of output for every line processed. When you use both options together, as in, for example:

```
lint -os( file.p ) -v1 -p file.c
```

then the single line will be preceded by the name of the file and the line number both enclosed in a C comment. This will enable you to track through every line of every header processed.

-parent(***Parent*****,** ***Child*** [**,** ***Child***] ... **)**

adds a link or links to the strong type hierarchy. See Section 9.6.3 Adding to the Natural Hierarchy

-passes(*k* [,*Options1* [,*Options2*]])
controls the number of passes lint makes over the source code. The default is 1. This is described in detail in Section 10.2.2.4 Specific Walk Options Summary.

+ppw(*word1* [, *word2*] ...) adds
-ppw(*word1* [, *word2*] ...) removes
PreProcessor command Word(s) ***word1***, ***word2***, etc. PC-lint/FlexeLint might stumble over strange preprocessor commands that your compiler happens to support (for example some Unix system compilers support **#ident**). Since this is something that canNOT be handled by a suitable **#define** of some identifier we have added the **+ppw** option. For example, **+ppw(ident**) will add the preprocessor command alluded to above. PC-lint/ FlexeLint then recognizes and ignores lines beginning with **#ident**.

If PC-lint/FlexeLint understands the semantics of the preprocessor word that is being enabled, the appropriate behavior will occur. See Section 15.4 Non-Standard Preprocessing See Section 5.8.6 In-line assembly code

--ppw(*word1* [, *word2*] ...)
This option removes any predefined meaning we may associate with the preprocessor word(s) (***word1***, ***word2*** etc.). If this is followed by a **+ppw(*word*)** the word is entered as a no-op rather than one that has a predefined meaning. For example, if your code contains the non-standard preprocessor directive **#dictionary** and if its meaning coincides with that of the DEC VMS compiler, then just issue the option **+ppw(dictionary)**. However, if you would rather have it ignore such a preprocessor word, issue the commands:

```
--ppw(dictionary)
+ppw(dictionary)
```

-ppw_asgn(*word1*, *word2*)
assigns the preprocessor semantics associated with ***word2*** to ***word1*** and activates ***word1***. E.g.

```
-ppw_asgn( header, include )
```

will then make

```
#header <stdio.h>
```

behave exactly like:

```
#include <stdio.h>
```

The purpose of this option is to support special non-standard preprocessor conventions provided by some given compiler.

Even though ***word2*** may not be activated it may still have semantics. Thus

```
-ppw_asgn( ASM, asm )
```

will assign the semantics associated with the **asm** preprocessor directive to **ASM** and activate **ASM**; all this in spite of the fact that **asm** has not been activated (with the **+ppw** option). See Section 15.4 Non-Standard Preprocessing for descriptions of non-standard preprocessing directives.

The **#macro** pre-processing directive is not a directive implemented by any compiler to our knowledge. So why, you ask, are we providing this directive? We are providing **#macro** as a way for programmers to implement arbitrary preprocessor directives by converting directives into macros.

For example, one compiler accepts the following preprocessor directive

```
#BYTE n = 'a'
```

as a declaration of the variable **n** having a type of **BYTE** and an initial value of **'a'**. The **#macro** directive will allow us to express **#BYTE** as a macro and so render the directive as C/C++ code.

The preprocessing directive:

```
#macro a b c
```

will result in the macro invocation

```
sharp_macro( a b c )
```

The word **sharp** is used as a prefix because the word '**sharp**' is often used to denote verbally the **'#'** character.

We can transfer the properties of **#macro** to some other actual or potential preprocessing directive using the option **ppw_asgn**. For example:

```
+ppw_asgn( BYTE, macro )
```

will assign the **#macro** properties to **BYTE** (and also enable **BYTE** as a preprocessing directive). Then the directive

```
#BYTE n = 'a'
```

will result in the macro call:

```
sharp_BYTE( n = 'a' )
```

Presumably there is a macro definition that resembles:

```
#define sharp_BYTE(s)  unsigned char s;
```

Such a definition can be placed in a header file that only PC-lint/FlexeLint will see by utilizing the **-header** option.

-printf(***N***, ***name1*** [, ***name2***] ... **)**

This option specifies that ***name1***, ***name2***, etc. are functions, which take **printf-**like formats. The format is provided in the ***Nth*** argument. For example, lint is preconfigured as if the following options were given:

```
-printf( 1, printf )
-printf( 2, sprintf, fprintf )
```

For such functions, the types and sizes of arguments starting with the variant portion of the argument list are expected to agree in size and type specified by the format. The variant portion of the argument list begins where the ellipsis is given in the function declaration. See also **-scanf**, **-wprintf**, **-wscanf** below and Section 11.1 Function Mimicry (-function).

Special Microsoft Windows option: If the number ***N*** is preceded by the letter **w**, pointers must be **far**. This is to support the Windows function **wsprintf**. The appropriate option then is:

```
-printf(w2,wsprintf)
```

-printf_code(***Code*** [**,** ***Type***] **)**

This option allows user-defined **printf** codes. For example, suppose a compiler's library allows **%TH** to be a special code for displaying **int** and allows **%$** to be a special code to display strings (nul-terminated). Moreover, suppose that the **%** (i.e. percent blank) has some special formatting function. The user's **printf** statement might look like:

```
printf( "x = %TH % %$\n", 37, "some string" );
```

Since this is not standard, PC-lint/FlexeLint will normally complain. To legitimize this, use the options:

```
-printf_code( TH, int )
-printf_code( $, char * )
-printf_code( " " )
```

Any sequence of characters except one involving the single or double quote can be used for the ***Code***. To avoid ambiguity you may quote the sequence as was done in the third example above. The absence of a ***Type*** (as in the third example) implies that the ***code*** is recognized and ignored and that there is no corresponding argument. The usual modifiers (length modifiers, zero fill, etc.) may be used in conjunction with the new ***Code***.

The ***Type*** is not evaluated until the format code is encountered so that user-defined types or typedef names may be used.

-printf_code affects the entire **printf** family of functions but does not set codes for **scanf** functions. For those use **-scanf_code**.

-restore restores the state of the error inhibition settings (see **-save**) to their state at the start of the last **-save**. For example:

```
/*lint -save -e621 */
  some code
 /*lint -restore */
```

temporarily suppresses Warning **621**. It is better to restore **621** this way than with a **+e621** because if **621** had been turned off globally (say, at the command line) this sequence will not accidentally turn it back on. **-restore** will also pop the most recent **-save** if any, so that **-save -restore** sequences can be nested. If no **-save** had been issued **-restore** restores back to the state at the beginning of the module.

Like **-save** there are two forms of the **-restore** option. An inner **-restore** is placed in a **/*** comment. An outer **-restore** is placed outside any module. An inner **-restore** will restore from the last inner **-save**. An outer **-restore** will save from the last outer **-save**. See **-save** for more details.

The restoration is effective when the next token is "scanned over". Thus

```
a  =  /*lint -save -e54 */   b / 0   /*lint -restore */;
```

will delay the restore until the '**;**' is "scanned over" (seeking the next token beyond '**;**'). This is a departure from earlier versions of our product (versions 6.00 and earlier) in which the restoration was done immediately. The reason for the change is that the parser must scan ahead by one token (in order to know, for example, what the full quotient of the division is in the above example). Hence the restoration would come sooner than was really appropriate. An immediate restore is still available using the option **-restore_** (the same option with one trailing underscore).

-restore can also be issued on the command line or in a **.lnt** file. See **-save**.

-restore_at_end has the effect of surrounding each source filename argument with an outer **-save** and **-restore**. (See **-save** option.) E.g., if we have two source modules, **t1.c** and **t2.c**, and a project file **t.lnt**:

```
t.lnt:
    -restore_at_end    // don't carry message suppression
                       // to subsequent modules
    t1.c
    t2.c

t1.c:
    /*lint -e54  */
    int i = 1/0;       // 54 not issued

t2.c:
    int j = 1/0;       // 54 issued
                       // the suppression from t1.c not
                       // fed forward because of +restore_at_end
```

then as indicated above although we suppress 54 within **t1.c** this will have no effect on **t2.c**

+rw(***word1*** [, ***word2***] ... **)** adds
-rw(***word1*** [, ***word2***] ... **)** removes
Reserved Word(s) ***word1***, ***word2***, etc. If the meaning of a reserved word being added is already known, that meaning is assumed. For example, **+rw(fortran)** will enable the reserved word **fortran**. If the reserved word has no prior known semantics, then it will be passed over when encountered in the source text. As another example:

```
+rw( __inline, entry )
```

adds the two reserved words shown. **__inline** is assigned a meaning consistent with that of the Microsoft C/C++ compiler (See Section 5.8.2 Compiler Codes). **entry** is assigned no meaning; it is simply skipped over when encountered in a source statement. Since no meaning is to be ascribed to **entry**, it could just as well have been assigned a null value as in

```
-dentry=
```

As a special case, if ***wordn*** is ***ms**, then all the Microsoft keywords are identified. Thus **+rw(*ms)** adds all the Microsoft keywords. This would not normally be necessary for Microsoft users since **co-msc*.lnt** has the **-cmsc** option embedded within it and this option also enables the Microsoft keywords. However, users of other compilers may wish to enable these keywords because they have become something of a de-facto standard.

By default, a number of Microsoft's keywords are pre-enabled in PC-lint because they are so commonly used. To deactivate all of them use **-rw(*ms)**. See Section 5.8.2 Compiler

Codes under **-cmsc** for the current list of supported Microsoft keywords (reserved words).

--rw(***word1*** [**,** ***word2***] ... **)**
removes preconceived notions as to what the potential reserved word(s) (***word1***, ***word2***, etc.) mean and gives it a default meaning (to ignore it). For example, **+rw(interrupt)** installs the reserved word **interrupt** with the meaning it has for the Microsoft compiler. If you don't want that meaning but would rather have **interrupt** ignored, then use the option sequence:

```
--rw(interrupt)
+rw(interrupt)
```

-rw_asgn(***word1, word2*** **)**
assigns the keyword semantics associated with ***word2*** to ***word1*** and activates ***word1***. E.g.

```
-rw_asgn( interrupt, _to_brackets )
```

will assign the semantics of **_to_brackets** to **interrupt**. This will have the effect of ignoring **interrupt(21)** in the following:

```
void f( int n ) interrupt(21) { }
```

The purpose of this option is to support special non-standard keyword conventions provided by some given compiler. But do not overlook the use of the **-d** option in this connection. **-d** (or the equivalent **#define**) can be more flexible since a number of tokens may be associated with a given identifier.

-save saves the current state of error inhibition settings. This state then can be restored with a **-restore** option.

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

```
-e#
+e#
-ealetter
-epletter
+/-efreeze
-w#
```

A **-save** option can be given within a module (with a **/*lint** comment) or outside a module. We call the first an inner **-save** and the latter an outer **-save**.

An inner **-save** can be used in a recursive option inhibition setting. For example,

```
#define alpha \
```

```
/*lint -save -e621 */ \
something \
/*lint -restore */
```

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

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

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

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

The outer **save**/**restore** facility saves and restores exactly the same error suppression parameters as the inner **save**/**restore** facility.

-save/**-restore** options that appear in source files are unrelated to **-save**/**-restore** options that appear outside of source files. Inside a module, it is impossible to **-restore** back to an error state that was saved outside the module.

An implicit outer **-save** occurs at the beginning of all processing; also, an implicit inner **-save** occurs at the beginning of processing for each module.

If you have more **-restore** options than **-save** options within a module, then the extra **-restore** options will revert the error state back to what was in effect at the beginning of processing for that module.

Similarly, if you have more **-restore** options than **-save** options outside of a module, then the extra **-restore** options will revert the error state back to what was in effect at the beginning of all processing. For PC-lint users, this means that you revert back to the "blank" state that you had before the **%LINT%** environment variable was read and before any command line options were processed. For FlexeLint users, the same is basically

true, except that the error state given by the **OPTIONS** macro in **custom.c** will be in effect.

+scanf(***N, name1*** [, ***name2***] ... **)**

This option specifies that ***name1***, ***name2***, etc. are functions, which take **scanf-**like formats. The format is provided in the ***Nth*** argument. For example, lint is preconfigured as if the following options were given:

```
-scanf( 1, scanf )
-scanf( 2, sscanf, fscanf )
```

For such functions, the types and sizes of arguments following the ***Nth*** argument are expected to be pointers to arguments that agree in size and type with the format specification. See also **-printf** above.

-scanf_code(***Code*** [**,** ***Type***] **)**

This option allows user-defined **scanf** codes. For example, suppose a compiler's library allows **%TH** to be a special code for reading **int** and allows **%$** to be a special code to read strings (nul-terminated). Moreover suppose that **%** (i.e. percent blank) is to be ignored. The user's **scanf** statement might look like:

```
int n;
char buffer[100];
scanf( "%TH % %$\n", &n, buffer );
```

Since this is not standard, PC-lint/FlexeLint will normally complain. To legitimize this, use the options:

```
-scanf_code( TH, int )
-scanf_code( $, char )
-scanf_code( " " )
```

Almost any sequence of characters can be used for the ***Code***. To avoid ambiguity, the code may be quoted as in the third example above. The usual modifiers (length modifiers, zero fill, etc.) may be used in conjunction with the new ***Code***.

It is assumed that each argument associated with a given ***Code*** is a pointer to the ***Type*** specified (or, as in the example above, an array that is convertible to a pointer). If the ***Type*** is missing, the format is recognized and ignored and no argument is matched against it.

The ***Type*** is not evaluated until the format code is encountered so that user-defined types or typedef names may be used.

-scanf_code affects the entire **scanf** family of functions but does not set codes for **printf** functions. For those use **-printf_code**.

-sem(*name* [,*semantic*] ...)

This option allows the user to endow his functions with user-defined semantics, or modify the pre-defined semantics of built-in functions. For example, the library function **mempy(*a1*,*a2*,*n*)** is pre-defined to have the following semantic checking. The third argument is checked to see that it does not exceed the size (in bytes) of the first or second argument. Also, the first and second arguments are checked to make sure they are not NULL.

To represent this semantic you could have used the option:

```
-sem( memcpy, 1P >= 3n && 2P >= 3n, 1p, 2p)
```

The details of semantic specifications are contained in Section 11.2 Semantic Specifications.

-setenv(*directive*)

will allow the user to set an environment string. The ***directive*** is of the form ***name=value***. For example:

```
-setenv(ROOT_DIR=\home\program\dev)
```

will set the environment variable **ROOT_DIR** to the indicated directory. This can be used subsequently in PC-lint/FlexeLint options by using the **%*var*%** syntax. For example:

```
-i%ROOT_DIR%\include
```

establishes a new search directory based on the environment name.

The environment variable setting will last for the duration of the process.

-size(*flags*, *amount*)

causes an Informational message (**812** or **813**) to be issued whenever a data variable's size equals or exceeds a given amount. Flags can be used to indicate the kind of data as follows:

s static data (Info **812**). Data can be file scope (**extern** or **static**) or declared **static** and local to a function.

a auto data (i.e., stack data) (Info **813**). See also **-stack** which can do a comprehensive stack analysis.

The purpose of the **-size** option is to detect potential causes of stack overflow (using the '**a**' flag) or to flag large contributors to excessively large static data areas.

E.g. **-size(a,100)** detects auto variables that equal or exceed 100 bytes. If you have a stack overflow problem, such a test will let you focus on a handful of functions that may

be causing the overflow. It does not, however, look at call chains and does not compute an overall stack requirement either of a single function or of a sequence of calls.

If ***amount*** is 0 (it is by defualt) no message is given

+source(***sub-option*****,...)**
-source(***sub-option*****,...)**

The **+source** form of this option will cause all source lines of a file or files to be echoed to the output stream. The purpose is to enable the programmer to see diagnostic messages in the context of a complete file.

The **-source** variant of this option will enable the user to specify ***sub-options*** without triggering the echoing. Presumably the **-source** option is placed in a **.lnt** file where it can house options in a dormant form which will not be triggered until the appearance of a **+source** on the command line.

When lines are displayed they are, by default, numbered. Also, by default, only module files (i.e. non-header files) are echoed. Through the use of sub-options, headers can be displayed with and without line numbers. Lines within headers are normally indented but this can be controlled through options.

The sub-options are:

-number -- do not number lines

-indent -- do not indent the lines within header files. Normally they are indented to the degree their inclusion is nested within headers.

+class(***identifier*** [**,** ***identifier***]...**)** -- specifies the set or sets of header files whose source will be echoed. This option is similar to **+libclass()**. Each ***identifier*** can be one of:

angle	All headers specified with angle brackets.
foreign	All headers found in directories that are on the search list (**-i** or **INCLUDE** as appropriate.
ansi	The 'standard' ANSI/ISO C header files, viz.
all	All header files
project	Project headers (i.e. non-library headers)

For example,

```
+source( +class(all) )
```

specifies that all header files will be echoed (as well as all module files). Also,

```
+source( +class(project) )
```

specifies that all non-library header files will be echoed (as well as all module files).

+dir(***directory*** [**,** ***directory***] ... **)** echo headers found in ***directory***
-dir(***directory*** [**,** ***directory***] ... **)** suppress the echo of headers found in ***directory***. If a ***directory*** is activated then all header files found within the directory will be echoed (unless specifically inhibited by the **-libh** option). It overrides the **+class** option for that particular directory. This option is similar to the way that the **libdir** option can be used to specify library headers. Wild card characters can be used.

+h(***file*** [**,** ***file***] **)** echo header ***file***
-h(***file*** [**,** ***file***] **)** suppress the echo of header***file***
This is similar to the way the **+libh** option can be used to designate that some headers are library. Wild card characters can be used.

-m(***file***, [**,** ***file***] **)** suppress the echo of specified modules. Wild card characters can be used.

-specific(***Options1***, [**,** ***Options2***] **)**
allows the user to specify options just before the start of a Specific Walk (***Options1***) and just after completion (***Options2***). See Section 10.2.2.4 Specific Walk Options Summary

-specific_climit(***n*** **)** specifies a Call limit. The calls recorded for any one function is limited to ***n***. See Section 10.2.2.4 Specific Walk Options Summary

-specific_wlimit(***n*** **)** specifies a Walk limit. At the end of the General Processing, any Specific Call that had not earlier been walked (in some prior pass) will result in a Specific Walk. See Section 10.2.2.4 Specific Walk Options Summary

-specific_retry(***n*** **)** indicates whether Specific Calls walked in one pass are rewalked on subsequent passes. ***n*** can be either 0, meaning no, or 1 meaning yes. The default is 1. See Section 10.2.2.4 Specific Walk Options Summary

-/+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. See Section 13.13 Stack Usage Report for more details and complete listing of the sub-options.

-static_depth(***n*** **)**
adjusts the depth of static variable analysis. By static variable, we mean any variable with static storage duration. These are variables declared at file scope (or, for C++, namespace scope) or any variable declared as static within a function. ***n*** corresponds to the number of call levels that each function will take into account in tracking external variables.

n=0 If ***n*** is 0, a function will not take into account any external variables.
n=1 If ***n*** is 1, then a function will track only externals that it uses.
n=2 If ***n*** is 2 or more, then a function will track external variables used by itself or by any function reachable in ***n***-1 calls.

Thus, if **alpha()** calls **beta()** and **beta()** calls **gamma()** and if the **static_depth** is 2 then the analysis of **alpha()** will include static variables used by **alpha()** and by **beta()** but not necessarily any others. If the **static_depth** is increased to 3 then the static variables used by **gamma()** are included in the value tracking of **alpha()**.

Thus, if the depth is high enough, all functions will track all variables that they could possibly affect. But increasing the depth will tend to slow the linting process.

By default the **static_depth** is 1.

See also **10.2.3 Tracking Static Variable** and flag **+fsv**.

-strong(***flags*** [**,** ***name***] ... **)**
identifies each ***name*** as a strong type with properties specified by ***flags***. Presumably there is a later **typedef** defining any such ***name*** to be a type. Strong types are completely described in Chapter 9. STRONG TYPES.

-subfile(*indirect-file***, options|modules)**
This is an unusual option and is meant for front-ends trying to achieve some special effect. There are two forms of the option; one with the second argument equal to **options** and the other with the second argument equal to **modules**. In general, indirect files (those ending in either **.vac** or **.lnt**) will contain both options and modules. Sometimes it is important to extract just the options from such a file. One example is if you are attempting to do a unit-check on one particular module. Say your project file is **project.lnt**. Then you might do project and unit checks using the same indirect file.

```
lint project.lnt             // project check
lint -subfile( project.lnt, options )  filename   // unit check
```

Note that **project.lnt** may itself have indirect files and that modules and options may be interspersed. The rule is that every indirect file is followed for as long as it takes until the first module is encountered. Every option thereafter is considered not a general option but specific to project check out.

With **modules** as the second argument to **subfile**, the processing picks up at precisely the point that the '**options**' subargument left off. Thus if you wanted to place a particular option, say **-e1706**, just before the first module of **project.lnt** you could achieve that effect by placing the following in either an indirect file or on the command line:

```
-subfile( project.lnt, options )
-e1706
```

```
-subfile( project.lnt, modules )
```

-summary([*output-filename*] **)**

This option causes a summary of all issued messages to be output after Global Wrap-up processing. If a filename is specified, the output is sent to the named file. If not, output is sent to the same output stream used for normal Lint messages (that is, the one specified by **-os()** or, if no such option was issued, **stdout**).

For each message issued, the summary information consists of the message number (e.g., **414** for "Possible division by 0"), the number of times that the message was issued, the message type (e.g., "Error", "Warning", etc.) and the message text. This forms a list of all Lint messages that were issued. The list is preceded by a row of column labels to aid readability.

The formatting of the output can be controlled by using the option:

```
-format_summary=...
```

which can be used like the **-format** option. The escape options usable with **-format** are also usable with **-format_summary**. The available format specifiers are:

.

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

The default summary format is:

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

Example: Suppose we have a source file called **"t.cpp"** which contains:

```
int a = 0;
int b = 0;
void g();
```

And suppose we have an indirect file **"a.lnt"** which contains:

```
-"format_summary=%c | %n | %t  --> %m"
-summary(s.txt)
t.cpp
```

And suppose we run:

```
lint a.lnt
```

Then our Lint output will be:

```
--- Module:   t.cpp (C++)

    _
void g();
t.cpp  3  Info 1717: empty prototype for function declaration, assumed
   '(void)'

--- Wrap-up for Module: t.cpp

Info 752: local declarator 'g(void)' (line 3, file t.cpp) not referenced
t.cpp  3  Info 830: Location cited in prior message

--- Global Wrap-up

Info 714: Symbol 'a' (line 1, file t.cpp) not referenced
t.cpp  1  Info 830: Location cited in prior message
Info 714: Symbol 'b' (line 2, file t.cpp) not referenced
t.cpp  2  Info 830: Location cited in prior message
```

And **s.txt** will contain:

```
Count | Number | Type  --> Text

2 | 714 | Info  --> Symbol '___' (___) not referenced
1 | 752 | Info  --> local declarator '___' (___) not referenced
1 | 1717 | Info  --> empty prototype for ___, assumed '(void)'
```

-t*#* sets PC-lint/FlexeLint's idea of what the Tab size is. This is used for indentation checking. By default PC-lint/FlexeLint presumes that tabs occur every 8 column positions. If your editor is converting blanks to tabs at some other exchange rate, then use this option. For example **-t4** indicates that a tab is worth 4 blank characters.

-tr_limit(***n*** **)**
allows the user to specify a Template Recursion limit. When the limit is reached, message **1777** is issued which reminds you that you may use this option to deepen the level of recursion. See message **1777** for further details.

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

```
class A{};
void g(A a) {}
// Lint reports "Info 1746: parameter 'a' in function 'g(A)'
// could be made const reference"
//lint +typename(174?)
void f(A a) {}
```

```
// Lint reports "Info 1746: parameter 'a' of type 'A' in function 'f(A)'
// of type 'void (A)' could be made const reference"
```

One of the purposes of this option is to show the user an exact type name to use as an argument to **-etype()**.

-u unit checkout -- This is one of the more frequently used options. It is used when linting a subset (frequently just one) of the modules comprising a program. **-u** suppresses the inter-module messages **526**, **552**, **628**, **714**, **729**, **755**-**759**, **765**, **768**, **769**, **974**, **948**, **1526**, **1527**, **1711**, **1714**, **1715**, and **1755**. Aside from this, there is no change in processing.

--u This option is like **-u** except that any module at a lower **.lnt** level is ignored. Suppose, for example, that **project.lnt** is a project file containing both options and module names. Then the command line:

```
lint --u project.lnt alpha.cpp
```

will do a unit check on module **alpha.cpp**. It will ignore any module names that may be identified within **project.lnt**. **project.lnt** does not have to immediately follow the **--u** option. Any **.lnt** file within **project.lnt** will similarly be processed for options but module names will be ignored. See also **-subfile()** which deals with this issue in a more comprehensive manner.

See also **-d**... and **+d**...

-unreachable indicates that a point in the program is unreachable. This is useful to inhibit some error messages. For example:

```
int f(n)
    {
    if(n) return n;
    exit(1);
    //lint -unreachable
    }
```

contains an unreachable indicator to prevent PC-lint/FlexeLint from thinking that an implied return exists at the end of the function. An implied return would not return a value but **f()** is declared as returning **int**.

-u*Name* can be used to Undefine an identifier that is normally pre-defined. For example:

```
-u_lint
```

will undefine the identifier **_lint**, which is normally pre-defined before each module. The undefine will take place for all subsequent modules after the default pre-definitions are established. If given within a lint comment, the undefine will take place immediately

as well as in subsequent modules (similar to **-d**...). The observant reader will notice that you may not undefine the name **nreachable**.

--u*Name* inhibits the macro ***Name*** from becoming defined. For example:

```
//lint --uX
#define X 1
int y = X;
```

will be equivalent to:

```
int y = X;
```

Please note the difference between this option and the **-u*Name*** option, which undefines any built-in definition for ***Name*** but does not affect definitions which ***Name*** may acquire in the future.

One example of how this option could be used is when macros hide machine dependent constructs. For example, a header file may define:

```
#define memcpy(a,b,n) { asm some-assembly required }
```

The option **--umemcpy** will cause Lint to ignore this definition. Often the **memcpy()** is declared as a function as well as defined as a macro and if this is the case, setting the option is all that would be required. At the worst you would have to introduce a declaration for the function. Declarations of this kind can be placed in a header file and automatically included within each module using the **-header** option.

-w*Level* This option allows the user to set a Warning level in a fashion similar to many compilers. The warning levels are:

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

The default warning level is level 3.

The option **-w*Level*** will establish a new warning level and affect only those messages in the "zone of transition". Thus, the option:

```
-e521  -e41  -w2
```

will have the effect of suppressing **521**, **41** and all Informationals. On the other hand

```
-e521  -e41  -w1  -w2
```

will suppress **41** and all Informationals. Warning **521** will be restored by the **-w2** because Warnings are in the zone of transition in going from level 1 to 2.

Because options are processed in order, the combined effect of the two options: **-w2 +e720** is to turn off all Informational messages except **720**.

-wlib(*Level*) sets a warning level for processing library headers and library modules. It will not affect C/C++ source modules. ***Level*** may have the same range of values as **-w*Level*** and are as follows:

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

A message encountered while processing a library header is subject to all the error inhibitors active normally and in addition is subject to suppression given by **-wlib** and **-elib**. It is for this reason that the default level is surprisingly high. **-wlib** can, generally, be rewritten as a sequence of **-elib** options. For example,

```
-wlib(2)
```

is equivalent to

```
-elib(7??) -elib(8??) -elib(9??)
-elib(17??) -elib(18??) -elib(19??)
```

but easier to type.

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

```
-wlib(1)
```

to inhibit all library messages except syntactic errors.

-wprintf(***N, name1*** [**,** ***name2***] ... **)**
This option is similar to the **-printf** except it is intended to be used with the wide-character variants of **printf**. In particular it is to be used with **wprintf**, **fwprintf** and **swprintf**. The property that makes **-wprintf** different from **-printf** is that the **%s** and **%c** formats assume wide characters.

-wscanf(*N, name1* [, *name2*] ...)

This option is similar to the **-scanf** except it is intended to be used with the wide-character variants of **scanf**. In particular it is to be used with **wscanf**, **fwscanf** and **swscanf**. The property that makes **-wscanf** different from **-scanf** is that the **%s** and **%c** formats assume wide characters.

+xml(*name*)

By adroit use of the **-format** option you may format output messages in xml. See the file **env-xml.lnt** for an example. This option has two purposes. Special xml characters ('**<**', '**>**' and '**&**' at this writing) will be escaped (to "**<**", "**>**" and "**&**" 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.

It is also possible to add a tag in angle brackets that could be used to define, for example, the version of **xml**. Thus:

```
+xml(?xml version="1.0" ?)
+xml(doc)
```

will produce as a prefix the following two lines.

```
<?xml version="1.0" ?>
<doc>
```

Then, at the end of all the message output the following one line will appear.

```
</doc>
```

-zero sets the exit code to **0**. This is useful to prohibit the premature termination of **make** files.

-zero(#) will set the exit code to zero if all reported errors are numbered # or higher after subtracting off 1000 if necessary. More precisely, messages which have a message number whose modulus 1000 is equal to or greater than #, do not increment the error count reported by the exit code. Note that suppressed errors also have no effect on the exit code. Use this option if you want to see warnings but proceed anyway.

-$** causes **$ to be regarded as an identifier character.

5.8 Compiler Adaptation

All compilers are slightly different owing largely to differences in libraries and preprocessor variables, if not to differences in the language processed. For PC-lint, the key to coping with

these differences is the selection and/or modification of one or two compiler-specific files provided on the distribution diskette.

There are two kinds of files. One has the form

```
co-xxx.lnt
```

This is used to specify lint options for a particular compiler. The second has the form

```
sl-xxx.c
```

and is used to describe the library for a K&R compiler (a compiler not using prototypes in its supplied header files). For ANSI/ISO compilers, just use the appropriate compiler options file as in

```
lint  co-xxx  module1  module2 ...
```

(the **.lnt** is understood). For K&R compilers use both files

```
lint  co-xxx  sl-xxx   module1   module2 ...
```

sl-*xxx*.c is an example of a "Library Module" as described in Section 6.2 Library Modules. You may safely modify either of these files. For example **co-*xxx*.lnt** effectively contains the option

```
-esym( 534, fclose )
```

This inhibits the Warning message (**534**) that you would otherwise get if you called **fclose** and did not check the return value for errors. But if your programming policy is to ***always*** check the return value of this function you could remove this option. Alternatively you can negate its effect with a **+esym(534, fclose)** (after the first option was issued).

Compiler specific options files are provided with PC-lint/FlexeLint and the most current files are available from our website.

If your compiler is not supported with a **co-*.lnt file**, you may want to modify **co.lnt**, which is the generic compiler options file. In addition, if your compiler does not provide prototypes and is not in the list above, you may wish to modify the file **sl.c**, which is a generic standard library file.

For FlexeLint, appropriate compiler options can be placed in a centrally located file whose extension is necessarily "**.lnt**" (See Section 4.1 Indirect (.lnt) Files). We refer to this file generically as **co.lnt**. See your FlexeLint Installation Notes and the subdirectory **/supp/** for any vendor-specific file of the form **co*.lnt**.

5.8.1 Microsoft Keywords

We use the phrases "Keywords" and "Reserved Words" interchangeably. They both refer to words that have special meaning at compile-time (or equivalently at lint-time). They are reserved in that they cannot be used for ordinary variable names or other user-supplied names. Thus the phrase "reserved word" is a little more honest even though "keyword" is more commonly used. In addition to the language-defined keywords, the Microsoft compilers have added over the years a number of non-standard keywords, which have become known as the ***Microsoft Keywords***. These are listed in Section 5.8.2 Compiler Codes under the option "**-cmsc**". The Microsoft keywords have become so prevalent in C/C++ compilers supporting operating systems following Microsoft standards that it is somewhat safer to include them than to exclude them. (Although see cautionary note below). Accordingly, PC-lint/FlexeLint will assume the Microsoft Keywords when hosted on one of the following operating systems:

Microsoft Windows
MS-DOS
OS/2

Note: this list does not include Unix or Linux!

It is possible to include or exclude these keywords as a group by using ***ms** as the reserved word ***name*** for the option **rw(*name*)**. Thus

```
-rw(*ms)
```

will remove these keywords. See the **-rw** option in Section 5.7 Other Options.

Cautionary Note: Even though the newer keywords added by Microsoft have a pair of leading '_' s, there is nonetheless a danger in adding too many of these keywords as you run the risk of collision with a variable name used within an inline function in some compiler header file. Ironically, header file writers have adopted the same leading double underscore convention to avoid collision with user macros. We will be diligent in attempting to find conflicts such as these and we will add appropriate **-rw** options to the appropriate **co-...lnt** files.

5.8.1.2 Additional Keywords

Keywords (i.e. reserved words) found below may be enabled using **+rw(*keyword*)**

__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**. The keyword **__packed** may appear before or after the keyword **struct**. For example,

```
struct __packed A { char a; int b; };
__packed struct B { char a; int b; };
```

Assuming the size and alignment of an **int** is 4 and the size of a **char** is 1 (default values) then in both cases the length of the **struct** will be 5 instead of the usual 8.

The keyword may also be used to indicate that a particular member does not require its ordinary alignment. For example:

```
struct C
    { char a; int __packed b; };
```

will cause member **b** to be located at offset 1 within the **struct.** Like **struct A** and struct **B**, **struct C** will also have a length of 5 rather than the usual 8.

5.8.2 Compiler Codes

For a number of compilers in the PC environment, there is a special option **-c*code*** (See Section 5.7 Other Options). For example, **-cmsc** selects the Microsoft C/C++ compiler. The **-c** option can serve to pre-define certain preprocessor symbols appropriate to the compiler according to the memory model selected. Otherwise, this would be awkward to do. It can also activate certain keywords.

If you are using one of the compiler options files named **co-*.lnt** then the appropriate **-c*code***, if any, is already set. For some systems there is no need for a **-c** option and none is provided. Instead, all necessary definitions are provided in a compiler options file.

The preprocessor symbols defined by **-c*code*** are in addition to the standard pre-defined symbols (described in Section 15.1 Preprocessor Symbols) and the symbol **_lint**, which is always pre-defined to be the version number without decimal point (such as 900). The verb "enable" below should be taken to mean "pre-define to the value normally defined by the designated compiler". This is intended to duplicate the preprocessing environment of your compiler. Please consult your compiler's documentation for further details.

If your compiler is not represented here or if we have missed a symbol, simply place the appropriate **-d** ... option in your compiler options file.

Option Compiler and Symbols

-caztec Manx Aztec C compiler.
Enables **MSDOS**, **__FUNC__** and **MPU8086** and, for large data pointers, **__LDATA**.

-cbc Borland C/C++
(same as **-ctc**)

-cc86 Computer Innovations C86
Enables **_C86_BIG** for large data pointers.

-cdesm DeSmet C
Enables the symbol **LARGE_CASE** for large data pointers and enables **#asm**.

-cdl Datalight C
Enables **MSDOS**, **DLC**, **I8086** and one of **I8086L**, **I8086D**, **I8086P** and **I8086S** depending on memory model. Also **LPTR** is defined to be 1 for large data models and **SPTR** is defined to be 1 for small data models.

-cec Ecosoft Eco-C88 (Version 4 and later)
Enables **__ECO**, and, for memory models having large data pointers, enables **__BIGDATA**, and, for memory models having large code pointers, enables **__BIGCODE**.

-cgnu GNU on Linux
This option is used for the GNU compiler. It ensures that the symbol **_LANGUAGE_C_PLUS_PLUS** is defined for C++ modules and the symbol **LANGUAGE_C** is defined for C modules. This is difficult to accomplish with just a compiler options file since compiler options files do not support conditionals. You would have to use **-header** and place a conditional within the header. This option avoids the complexity.

The use of this option also allows a number of features unique to the GNU compiler. One is indexed initialization of arrays. For example:

```
int a[100] = { [99] = -1 };
```

is a way of initializing the last element of an array without specifying the first 99 values.

Another feature is the use of variable macro arguments. For example:

```
#define P(f,a...) fprintf( stderr, f , ##a )
```

Then macro **P()** can be invoked with zero or more arguments after the format. Note that the space before the '**,**' before the '**##**' is critical to get the full effect.

Other features include the **#include_next** preprocessor command and the **__null** keyword.

-chc MetaWare High C/C++
Enables **MSDOS** and pre-assigns 0 to **_stdio_defs_included**, **__1167**, **_HIGHC_** and enables **#c_include**. As with the other compilers, Microsoft keywords are enabled.

-cht Hi-Tech C
pre-assigns 0 to symbols **z80** and **m68k** and pre-assigns 1 to symbol **i8086**.

-cibm32 IBM 32 bit for OS/2
This option (**-cibm32**) can be used to specify the 32 bit IBM compilers for OS/2. These include C Set 2 and Visual Age. The necessity for the **-c** option in this case is to support the preprocessor variables **__IBMCPP__** and **__IBMC__** depending on whether the module being processed is respectively C++ or C. This is, in fact, the only effect that **-cibm32** has. All other distinctive aspects of these compilers are covered in the file **co-ibm32.lnt**.

-cicb Intel Code Builder
pre-assigns variables **LINT_ARGS**, **_TIMESTAMP_**, **CH_TIME**, **_INTELC32_**, **_ARCHITECTURE_** (to **386**) and sets the size of **int** and **pointer** to 4 bytes and allows '**$**' in identifiers.

-clc Lattice C, Versions 2 and 3
MSDOS is enabled. **SPTR** is defined to be 1 if the memory model uses small data pointers and 0 otherwise. **LPTR** is defined to be 1 if the memory model uses large data pointers and 0 otherwise. In addition, **CPM80**, **CPM86** **CPM68**, **LATTICE**, and **I8086** are enabled and one of **I8086L**, **I8086D**, **I8086P** or **I8086S** depending on memory model.

-clc6 Lattice C, Version 6
This is like **-clc** except that, in addition, preprocessor words **ANSI**, **NULL** and **LC60** are enabled and those of the form **CPM*** are not. The user should enable **DOS**, **FAMILY** or **OS2** if using these symbols. Also, keywords **align**, **critical**, **noalign**, **nopad**, **pad**, **private**, **actual**, **inline** and their double underscore prefix versions are enabled.

-cmsc Microsoft C/C++, Versions 4.0 - 12.0(VC6)
This enables **MSDOS**, **M_I86** and one of **M_I86LM**, **M_I86CM**, **M_I86MM** and **M_I86SM** depending on memory model. The following reserved words called *Microsoft keywords* are added: **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, **_asm**, **_based**, **_cdecl**, **_export**, **_far**, **_fastcall**, **_fortran**, **_huge**, **_interrupt**, **_loadds**, **_near**, **_pascal**, **_saveregs**, **_segment**, **_segname**, **_self** **__asm**, **__based**, **__cdecl**, **__except**, **__export**, **__far**, **__fastcall**, **__fortran**, **__huge**, **__inline**, **__interrupt**, **__loadds**, **__near**, **__pascal**, **__saveregs**, **__segment**, **__segname**, **__self**, **__stdcall**, **__syscall**, **__try**. The **//** form of comment is understood for C programs. If **-A** is set then **NO_EXT_KEYS** is enabled and then special keywords and comment control are disabled.

-cmwc Mark Williams C
Enables **DOS_H** and, for large data pointers, **_LARGE**.

-csc Symantec C/C++

They enable **MSDOS**, **M_I86** and one of **M_I86LM**, **M_I86CM**, **M_I86MM** and **M_I86SM** depending on memory model. In addition to the Microsoft keywords the following keywords are added **__import**, **__ss**, **__handle** and **__iob**, which have no semantics.

-ctc Turbo C/C++ and Borland C/C++
__BORLANDC__, **__TURBOC__**, **__MSDOS__**, **__CDECL__** are enabled and one of **__LARGE__**, **__COMPACT__**, **__MEDIUM__**, **__SMALL__** are defined (to be 1) according to the memory model selected. For the large model, pointer differences are assumed to be **long**.

Additional keywords supported are: **asm**, **_asm**, **__asm**, **_ss**, **_es**, **_ds**, **_far16**, **__far16**, **_syscall**, **_cs**, **_export**, **_saveregs**, **_loadds**, **_seg** and **__seg**, and the Microsoft keywords. Please note that the keywords **_asm** (and **__asm**) obtained with **-ctc** (and **-cbc**) have different semantics than do the Microsoft keywords by the same name.

Register keywords (as in Turbo C, these are pre-declared to be of type **unsigned** or **unsigned char** depending on whether the associated register is 16 bits or 8 bits long). **_AX _BX _CX _DX _SI _DI _SP _BP _AH _AL _BH _BL _CH _CL _DH _DL _DS _ES _RS _SS _FLAGS**

-ctsc TopSpeed C
Enables **M_I86LM**, **M_I86CM**, **M_I86MM**, **M_I86SM** according to memory model (like Microsoft C).

-cwc Watcom C/C++
Enables **__WATCOMC__**, **MSDOS** and one of **M_I86LM**, **M_I86CM**, **M_I86MM** and **M_I86SM** depending on memory model. If **-A** is set, then **NO_EXT_KEYS** is enabled. **long char** is enabled (**+flc**). As with all compilers, the Microsoft keywords are also enabled.

-cwh Whitesmith C
Establishes **@** as a one-character identifier. Thus: **@abc** consists of two identifiers "**@**" and "**abc**". A one-character identifier is a unique lexical unit. It differs from a regular identifier character such as a letter in that it does not join with other characters and it differs from a special character (like comma) in that it can be used as the name of a macro. In this capacity, **@** is pre-defined to be **_gobble** (See Section 5.8.3 Customization Facilities). In effect, PC-lint/FlexeLint ignores **@** and the token which follows. The same effect can be achieved with the options:

```
-ident1(@)
+rw(_gobble)
-d"@=_gobble "
```

See Section 5.8.3 Customization Facilities for an alternative treatment of **@**

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.

5.8.4 Identifier Characters

Additional identifier characters can be established. See **-$** and **-ident()** in Section 5.7 Other Options.

5.8.5 Preprocessor Statements

See Section 15.4 Non-Standard Preprocessing for special non-standard preprocessor statements. Also see the **+ppw** option in Section 5.7 Other Options.

5.8.6 In-line assembly code

Compiler writers have shown no dearth of creativity in their invention of new syntax to support assembly language.

In the PC world, the most frequently used convention is (to simplify slightly):

asm { *assembly-code* }

or

```
asm assembly-code <new-line>
```

where **asm** is sometimes replaced with either **_asm** or **__asm**. This convention is supported automatically by enabling the **asm** keyword, using **+rw(asm)** (or **+rw(_asm)** or **+rw(__asm)** as the case may be).

But other conventions exist as well. One manufacturer uses

```
#asm
    assembly-code
#
```

For this sequence, it is necessary to enable **asm** as a pre-processor word using **+ppw(asm)**

If your compiler uses a different preprocessor word, you may use the option **+ppw_asgn**. See Section 5.7 Other Options

Another sequence is:

```
#asm
    assembly-code
#endasm
```

For this **+ppw(asm,endasm)** is needed.

Yet another convention is:

```
#pragma asm
    assembly-code
#pragma endasm
```

For this you need to define the two pragmas with

```
+pragma(asm, off)
+pragma(endasm, on)
```

5.8.7 Pragmas

5.8.7.1 Built-in pragmas

A number of compilers support the **push_macro** and the **pop_macro** pragmas.

push_macro(***name-in-quotes***), where the ***name-in-quotes*** specifies a macro, is a pragma that will save the definition of the macro onto a stack. This will allow the macro to be redefined

or undefined over a sub-portion of a module. Presumably this will be followed by a **pop_macro**(***name-in-quotes***) pragma that will restore the original macro. Thus:

```
#define N 100
#pragma push_macro( "N" )
#define N 1000
int x[N];
#pragma pop_macro( "N" )
int y[N];
```

declares **x** to be an array of 1000 integers whereas **y** becomes an array of 100 integers.

Note that you have, in effect, ***k*** different stacks where ***k*** is the number of different names provided as arguments to these pragmas.

5.8.7.2 User pragmas

+pragma(***identifier, action*** **)** adds a **pragma**
-pragma(***identifier*** **)** removes a **pragma**

The **+pragma(** ***identifier, action*** **)** option can be used to specify an ***identifier*** that will be used to trigger an ***action*** when the ***identifier*** appears as the first identifier of a **#pragma** statement. ***action*** must be one of

```
on
off
once
message
ppw
macro
fmacro
options
```

Please note that the purpose of the **+pragma** option is compatibility with your compiler. If your goal is to conditionally compile depending on the presence of PC-lint/FlexeLint, use the **_lint** preprocessor variable.

on and **off** An **off** action will turn processing off. An **on** option will reset processing. For example, assume that the following coding sequence appears in a user program:

```
#pragma ASM
    movereg 3,8(4)
#pragma ENDASM
```

Lint will normally ignore the **#pragma** statements but it will not ignore the assembly language between the **#pragma** statements, which might lead to a flurry of messages. To resolve the problem, add the pair of options:

```
+pragma( ASM, off )
+pragma( ENDASM, on )
```

This will turn off lint processing when the **ASM** is seen and turn it back on when the **ENDASM** is seen.

Please don't get this backwards. See Section 5.8.6 In-line assembly code

once The option **+pragma(*identifier*, once)** allows the programmer to establish an arbitrary ***identifier*** (usually the identifier **once**) as an indicator that the header is to be included just once. To mimic the Microsoft C++ compiler use the option: **+pragma(once,once)**. Then, a subsequent appearance of the pragma:

```
#pragma once
```

within a header will cause that header to be included just once. Subsequent attempts to include the header within the same module will be ignored.

Note: the **once** pragma is automatically enabled when the **-cmsc** option is given.

message The option **+pragma(*identifier*, message)** allows the programmer to establish an arbitrary ***identifier*** (usually the identifier **message**) as a pragma that will produce a message to standard out. For example:

```
+pragma(message,message)
```

will cause the pragma:

```
#pragma message "hello from file" __FILE__
```

to write to standard out a greeting identifying the file within which the pragma is contained. As this example shows, macros are expanded as encountered in the message line. Also, messages fall under control of the conditional compilation statements, **#if**, etc. Following the Microsoft compiler, if the first token is a left parenthesis then only the parenthetical expression will be output. Other information on the line is not output.

Note: the **message** pragma is automatically enabled when the **-cmsc** option is given.

ppw The option **+pragma(*indentifier*, ppw)** will endow ***identifier*** with the **ppw** pragma action which means reprocess the line as a preprocessor statement but with the word "**pragma**" ignored. Thus:

```
//lint +pragma( include, ppw )
#pragma include "abc.h"
```

will give the pragma keyword "**include**" the pragma action **ppw** which will cause the next line to operate just like a standard **#include** preprocessor line.

macro, **fmacro** and **options**

Pragmas provide an avenue for the programmer to communicate to a compiler that is not governed by the syntax of the language. In most cases PC-lint/FlexeLint can ignore this information. Occasionally, however, the programmer will want us to act on this information so that he, the programmer, is not inserting the information twice, once for his compiler and once for PC-lint/FlexeLint.

Obviously it is impossible to provide compatible pragma recognition for all compilers now and into the future. The best general method of handling this seems to be to convert the pragma into a macro. Through the macro definition process, the macro can then become whatever the programmer wants. In particular it can become a **/*lint** ***options***...***/** comment and thereby be converted into a PC-lint/FlexeLint option. Alternatively it can be converted into code.

There are three basic ways of converting a pragma into a macro; these are identified as pragma types **macro**, **fmacro** and **options**.

macro If a pragma is identified as **macro** as in the option

```
+pragma( identifier, macro )
```

then when a pragma by that name is encountered, the name is prefixed with the string '**pragma_**' and all the information to the right of the ***identifier*** is enclosed in parentheses. For example, one compiler supports statements such as:

```
#pragma port x @ 0x100
```

This would identify **x** as a particular I/O port. Later in the program there may be assignments to or from **x**. If PC-lint/FlexeLint were to ignore the pragma it would have to emit syntax errors on every use of **x**.

If the option **+pragma(port,macro)** is given, the above pragma will be converted into:

```
pragma_port( x @ 0x100 )
```

Presumably there is a macro definition that resembles:

```
#define pragma_port( s ) volatile unsigned s;
```

Such a definition can be placed in a header file that only PC-lint/FlexeLint will see by utilizing the **-header** option.

fmacro The second form of macroizing a pragma is identified as **fmacro** (meaning function macro). This would be used in a **+pragma** option having the form:

```
+pragma( identifier, fmacro )
```

With the **fmacro** type, instances of the pragma are assumed to already be in functional notation. Like the **macro** type, the name of the pragma is prefixed with **pragma_** to avoid conflicts with other uses of the pragma name. Aside from this prefixing the pragma is taken as found in the pragma statement and employed as a macro. For example, consider a pragma called **warnings** which appears as:

```
#pragma warnings(no)
```

or

```
#pragma warnings(yes)
```

Presumably the **no** form turns **off** warnings and the yes form turns them back on.

If the option **+pragma(warnings,fmacro)** is given, the first pragma will be converted into:

```
pragma_warnings(no)
```

and the second will be converted into:

```
pragma_warnings(yes)
```

The programmer will want to provide a set of macros that can convert these pragmas into the equivalent form for PC-lint/FlexeLint. This could be done as follows:

```
#define pragma_warnings(x)  pragma_warnings_##x
#define pragma_warnings_no  /*lint -save -w1 */
#define pragma_warnings_yes /*lint -restore */
```

These macro definitions can be placed in a header file that only PC-lint/FlexeLint will see by utilizing the **-header** option.

options The third form of pragma that can be macroized is identified as '**options**' using an option of the form:

```
+pragma( identifier, options )
```

When a pragma having the name ***identifier*** is used, it is presumably followed by a blank-separated sequence of options having the form ***name=value***. An example is :

```
#pragma OPTIONS tab=4 length=80
```

In this form, each individual option becomes a potential macro invocation. The name of this macro formed by concatenating **pragma_**, the ***indentifier*** which, in this case, is **OPTIONS**, followed by underscore and the name of the suboption. Thus for the suboption **tab** the name of the macro is **pragma_OPTIONS_tab**. As an example, suppose the option **+pragma(OPTIONS, options)** is given and the following macro defined.

```
#define pragma_OPTIONS_tab(x)  /*lint -t##x */
```

Then the above pragma will result in just the single macro invocation:

```
pragma_OPTIONS_tab(x)
```

This will invoke the tab option of PC-lint/FlexeLint. The '**length=80**' option is ignored. To attach meaning to the length option it would only be necessary to define a macro whose name would be **pragma_OPTIONS_length**.

identifier This option **-pragma(*identifier*)** will remove the pragma whose name is ***identifier***. Thus:

```
-cmsc
-pragma(message)
```

will remove the **message** pragma while using Microsoft C/C++.

5.8.8 The General Solution

If none of these special features resolves your particular problem, what can you do? You can always normalize the syntax by using a macro. For example, if a specialized robot controller allowed commands like

```
void f()
    {
    move down;
    move up;
    move clockwise;
    }
```

you can convert these into macros yielding:

```
void  f()
    {
    MOVE( down );
    MOVE( up );
    MOVE( clockwise );
```

```
}
```

This latter C code with the appropriate macro definition can then be taken anywhere. It can be fed to any compiler, C tool, or lint analysis tool given the appropriate definitions. For example, for PC-lint/FlexeLint we may have:

```
#ifdef _lint
#define MOVE(x) move( #x )
void move( char * );
#else
#define MOVE(x) move x
#endif
```

5.9 Self-Referencing Options Files

An options file such as **std.lnt** can include a reference to **std.lnt** and this is understood to mean the next file by that name in the search directory sequence.

The following example will motivate this feature. Suppose you have two different **lin.bat** commands which refer to two different versions of PC-lint. Say **lin8.bat** contains

```
\lint8\lint-nt.exe -i\lint8  std.lnt ...
```

and say that **lin9.bat** contains

```
\lint9\lint-nt.exe -i\lint9  std.lnt ...
```

Now suppose directory **\alpha** contains a project to be linted. So the programmer places in that directory a file whose name is **std.lnt** which will contain all the options unique to the **alpha project**. Then when we are linting out of directory **\alpha** the **std.lnt** peculiar to **\alpha** will automatically be found ahead of the **std.lnt** that appears in the **\lint** directories. The information within the project **std.lnt** is often intended to be a supplement to whatever information is contained in the **std.lnt** within the **\lint** directories. How might we include that? If there were only one **\lint** directory then the **std.lnt** that appears in **\alpha** could refer to **\lint\std.lnt** via an absolute reference. However, we have here assumed that there are two versions of **lin.bat** (at least). When **lin8.bat** is running we want to include **\lint8\std.lnt** and when **lin9.bat** is running we want to include **\lint9\std.lnt**.

Happily, all the programmer has to do is place a (recursive) reference to **std.lnt** in **std.lnt**. For example his **std.lnt** can contain:

```
std.lnt
-esym(512,alpha)
```

When his **std.lnt** is invoked the first thing that is seen is "**std.lnt**" which cannot be a self-reference and so must necessarily be the next **std.lnt** on the chain (**\lint8\std.lnt** or **\lint9\std.lnt**) depending on the **-i** options. After this is processed, control returns to process the **-esym** option.

6. LIBRARIES

Please note: This chapter is ***not*** about how to include header files that may be in some directory other than the current directory. For that information see the **-i** option (Section 5.7 Other Options) or Section 15.2.1 INCLUDE Environment Variable. This chapter explains how information in header files (and possibly modules) is interpreted.

Examples of libraries are compiler libraries such as the standard I/O library, and third-party libraries such as windowing libraries, and database libraries. Also, an individual programmer may choose to organize a part of his own code into one or more libraries if it is to be used in more than one application. The important features of libraries, in so far as linting is concerned, are:

(a) The source code is usually not available for linting.

(b) The library is used by programs other than the one you are linting.

Therefore, to produce a full and complete analysis it is essential to know which headers represent libraries. It is also possible for modules to be available for linting but, because they are created beyond the control of the immediate programmer, they too can benefit from the designation 'library'.

6.1 Library Header Files

A library header file is a header file that describes (in whole or in part) the interface to a library.

The most familiar example of a library header file is **stdio.h**. Consider the file **hello.c**:

```
#include <stdio.h>

main()
    {
    printf( "hello world\n" );
    }
```

Without the header file, PC-lint/FlexeLint would complain that **printf** was neither declared (Informational **718**) nor defined (Warning **526**). (The distinction between a declaration and a definition is extremely important in C/C++. A definition for a function, for example, uses curly braces and there can be only one of them for any given function. Conversely, a declaration for a function ends with a semi-colon, is simply descriptive, and there can be more than one).

If **hello.c** were a C++ program an even stronger message (Error **1055** instead of Informational **718**) would be issued, but we will assume a straight C program.

With the inclusion of **stdio.h** (assuming **stdio.h** contains a declaration for **printf**), PC-lint/FlexeLint will not issue message **718** (or the **1055** for C++). Moreover, if **stdio.h** is recognized as a library header file, (it is by default because it was specified with angle brackets), PC-lint/FlexeLint will understand that source code for **printf** is not necessarily available, see clause (a) above, and will not issue warning **526** either. Note: Other messages associated with library headers are not suppressed automatically. But you may use **-wlib** or any of the **-elib**... options for this purpose. See Section 5.2 Error Inhibition Options

Because of clause (b) above, not all components of a library header file need to be fully utilized over the course of compiling a program. Such components include: declared data objects and functions, types specified with **typedef**, macros specified with **#define**, **struct**, **union**, **enum** and **template** declarations and their members. For these components, messages **749**-**770** are suppressed. See Section 13.8 Weak Definials

A header file can become a library header file if:

(i) It falls within one of the four broad categories of the option **+libclass**, viz. **all**, **ansi**, **angle** and **foreign** (described below), and is not excluded by either the **-libdir** or the **-libh** option.

(ii) OR, for finer control, it comes from a directory specified with **+libdir** and is not specifically excluded with **-libh**.

(iii) OR, for the finest control, it is specifically included by name via **+libh**.

(iv) OR, is included within a library header file.

You may determine whether header files are library header files by using some variation of the **-vf** verbosity option. For each included library header you will receive a message similar to:

```
Including file c:\compiler\stdio.h (library)
```

The tag: '**(library)**' indicates a library header file. Other header files will not have that tag.

What follows is a more complete description of the three options used to specify if or when a header file is a library header file.

- **+libclass(** ***identifier*** [**,** ***identifier***] ... **)**
  specifies the set or sets of header files that are assumed to be library header files. Each ***identifier*** can be one of:

  angle All headers specified with angle brackets.

  foreign All header files found in directories that are on the search list (**-i** or **INCLUDE** as appropriate).

Thus, if the **#include** contains a complete path name then the header file is not considered '**foreign**'. To endow such a file with the library header property use either the **+libh** option or angle brackets. For example, if you have

```
#include "\include\graph.h"
```

and you want this header to be regarded as a library header use angle brackets as in:

```
#include <\include\graph.h>
```

or use the option:

```
+libh(\include\graph.h)
```

Similar remarks can be made about

```
#include "include\graph.h"
```

If a search list (specified with **-i** option or **INCLUDE**) is used to locate this file it is considered foreign; otherwise it is not.

ansi The 'standard' ANSI/ISO C header files, viz.

assert.h	**limits.h**	**stddef.h**
ctype.h	**locale.h**	**stdio.h**
errno.h	**math.h**	**stdlib.h**
float.h	**setjmp.h**	**string.h**
fstream.h	**signal.h**	**strstream.h**
iostream.h	**stdarg.h**	**time.h**

all All header files are regarded as being library headers.

By default, **+libclass(angle,foreign)** is in effect. This option is not cumulative. Any **+libclass** option completely erases the effect of previous **+libclass** options. To specify no class use the option **+libclass()**.

- **+libdir(** ***directory*** [**,** ***directory***] ... **)** activates
  -libdir(***directory*** [**,** ***directory***] ... **)** deactivates
  the directory (or directories) specified. The notion of ***directory*** here is identical to that in the **-i** option. If a ***directory*** is activated then all header files found within the directory will be regarded as library header files (unless specifically inhibited by the **-libh** option). It overrides the **+libclass** option for that particular directory. For example:

```
+libclass()
```

```
+libdir( c:\compiler )
+libh( os.h )
```

requests that no header files be regarded as library files except those coming from directory **c:\compiler** and the header **os.h** (see below). Also,

```
+libclass( foreign )
-libdir( headers )
```

requests that all headers coming from any foreign directory except the directory specified by **headers** should be regarded as library headers.

Wild card characters '*****' and '**?**' are supported.

Note: A file specified as

```
#include "c:\compiler\i.h"
```

is not regarded as being a library header even though **+libdir(c:\compiler**) was specified. Only files found in **c:\compiler** via a search list (**-i** or **INCLUDE**) are so regarded and only when the **-i** option matches the **libdir** parameter. For example,

```
#include "compiler\i.h"
```

will also not be considered as library even though the **-ic:** option is given, and the file is found by searching. The **-i** search directory (**c:**) is not matching the **libdir** directory (**c:\compiler**).

- **+libh(** ***file*** [**,** ***file***] ... **)** adds
  -libh(***file*** [**,** ***file***] ... **)** removes
  files from the set that would otherwise be determined from the **+libclass** and **+/-libdir** options. For example:

```
+libclass( ansi, angle )
+libh( windows.h, graphics.h )
+libh( os.h )   -libh( float.h )
```

requests that the header files described as **ansi** or **angle** (except for **float.h**) and the individual header files: **windows.h**, **graphics.h** and **os.h** (even if not specified with angle brackets) will be taken to be library header files.

Wild card characters '*****' and '**?**' are supported.

For **libh** to have an effect, its argument must match the string between quotes or angle brackets in the **#include** line. Thus in the case of:

```
#include <../lib/graphics.h>
```

you must have **+libh(../lib/graphics.h)**.

Note that the **libh** option is accumulative whereas the **libclass** option overrides any previous **libclass** option including the default.

When a **#include** statement is encountered, the name that follows the **#include** is defined to be the ***header-name*** (even if the name is a compound name containing directories). When an attempt is made to open the file, a list of directories is consulted, which are all those specified by **-i** options and the **INCLUDE** environment variable. The directory that is used to successfully open the file is defined to be the ***header-directory***.

The options **+libdir(...)** and **-libdir(...)** are applied to the ***header-directory*** and the options **+libh(...)** and **-libh(...)** are applied to the ***header-name*** (as defined in the previous paragraph). For example, given the following:

```
#include "graphics\shapes.h"
```

Suppose that the following option had been given:

```
-iC:\
```

and suppose further that a file **"C:\graphics\shapes.h"** exists. Then the ***header-name*** would be **"graphics\shapes.h"** and the ***header-directory*** would be **"C:\"**. Any one of the following options could be used to designate the file as a library file.

```
+libh( graphics\* )
+libh( *shapes.h )
+libdir( C:* )
+libdir( C:\ )
```

6.2 Library Modules

You may designate that a module is a library module using the option:

```
+libm( module-name )
```

You would normally use just the '**+**' form of the option. But you may use **-libm** to undo the effects of a **+libm** option with some arguments.

This option has the effect of designating the entire module and all of the header files that it includes, as "library". That is, messages will be inhibited via **-wlib** or **-elib**... options. Unused globals defined within such a module will draw no complaints, etc.

As an example suppose you have an application **alpha.c**, and that this code requires the services of a module **beta.c** which is generated by a separate program. Typically the interface to **beta.c** will be described by a header file **beta.h** and a typical linting can be specified by:

```
lint  +libh( beta.h )  alpha.c
```

But another possibility is to include **beta.c** in the lint. This would have the advantage of facilitating inter module value tracking. The typical command to do this would be:

```
lint  +libm( beta.c )  alpha.c beta.c
```

Note that by making **beta.c** "library" you are also making **beta.h** "library" (assuming the former includes the latter).

Note that the option **libm** takes a pattern that may include wild-cards. Let us suppose that our generator will generate not just **beta.c** but a sequence of three modules

```
beta1.c   beta2.c   beta3.c
```

Then they can all be designated as library with the single option

```
+libm( beta*.c )
```

6.3 Library Object Modules

If you have source code for a library, an alternative procedure (alternative to producing a library module as in the previous section), is to create a lint object module directly (See Chapter 8. LINT OBJECT MODULES). Assuming we have the same modules **g1.c**, **g2.c**, ... **g25.c** as in the preceding section, create the file **g.lnt** containing:

```
-u
-library g1.c
-library g2.c

   ...
-library g25.c
-oo(glib.lob)
```

Then issue the command

```
lint g.lnt
```

The resulting object module, **glib.lob**, may be used in conjunction with other modules as in:

```
lint glib.lob program.c
```

The advantage of this approach is that diagnostic information will be directed to the precise location within the original library source. You may or may not also wish to produce a **glib.h** file.

See Chapter 8. LINT OBJECT MODULES.

6.4 Assembly Language Modules

In this section we deal with the case of assembly-language modules. For in-line assembly code see Section 5.8.6 In-line assembly code.

If one or more modules of your application are written in assembly language or, equivalently, in some language other than C or C++ (a common phrase is "mixed language"), you must arrange so that the missing code does not cause PC-lint/FlexeLint to give spurious messages. The most common way of proceeding is to create a header file describing the assembly language portion of your application. This header file, say **asm.h**, will have property (a) of library header files in that the objects declared therein will not be defined in files seen by PC-lint/FlexeLint. Hence we make it a library header file with the option:

```
+libh(asm.h)
```

The header file will usually NOT have property (b); i.e. it will usually not be used by other programs. However, the usual weak definials such as macros and **typedef**, will not (or should not) appear in abundance in such a file. The worst that would happen in any case, is that some component of **asm.h** would escape use and you would not know about it. You can always remove the option that indicates **asm.h** is a library header and see what turns up.

Finally, the assembly language portion of your application may be the only portion of your application which is referencing, initializing or accessing some variable or function. A spurious "not referenced' or "not accessed" message would be given. The easiest thing to do is to explicitly suppress the message(s). For example, if the assembly language portion is the only portion accessing variable **alpha** and you are getting message **552**, then place option **-esym(552,alpha)** among your lint options. If you are using our suggested setup, as described in Section 16.2 Recommended Setup, then **std.lnt** will now have the contents:

```
c.lnt
options.lnt
+libh(asm.h)
-esym(552,alpha)  //accessed in assembly language
```

You might be tempted to place these options in lint comments within **asm.h**. Unfortunately, the **libh** option will be set too late to establish **asm.h** as a library header in the first module that

includes it (subsequent modules will be OK) and the **-esym** option would be lost if converting to object modules.

You might yet say that "My assembly language routines are sometimes opted out and sometimes opted in, and this is under control of a global preprocessor variable **USEASM**. When opted out, C/C++ equivalent routines are activated. How can I cope with this varying situation?"

This actually makes the situation easier. Just make sure that when you are linting, **USEASM** is opted out. You might use:

```
#ifdef _lint
#undef USEASM
#endif
```

or some equivalent sequence. In this way, lint will know the intent of the assembly code from the equivalent C/C++ code. The previously suggested options of **+libh** and **-esym** are then not necessary.

If these relatively simple solutions are not quite adequate then, for the ultimate in flexibility, do the following. Create a Library Module, **asm.l** for example. We use the '**l**' extension to denote that it is seen only by lint, not by the compiler. It can contain, for starters:

```
/*lint  -library  */
#if USEASM
#include "asm.h"
extern double alpha;
#endif
```

Within such a Library Module, we can check for preprocessor variables such as **USEASM** and react accordingly. Note that the declaration of **alpha** within a Library Module makes it a library symbol, and warnings about it not being accessed or referenced are inhibited.

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.

7.1.3 Monitoring pre-compiled headers

The sequence of events that takes place when a pre-compiled header is included can be monitored by using a variant of the verbosity option that contains or implies the letter '**f**'. Given the option sequence:

```
-pch(x.h) -vf
```

we would expect to see, at the first time **x.h** is included, the verbosity line:

```
Including file x.h (bypass)
```

As indicated above, **x.h** becomes, of necessity, a file to be bypassed in subsequent modules. After fully processing **x.h** and all of its includes we will see the line:

```
Outputting to file x.lph
```

The extension "**lph**" stands for "lint pre-compiled header". The name of the file containing the pre-compiled output is formed by appending this extension onto the root of the file named in the **pch** option.

In subsequent modules you will see the verbosity line:

```
Bypassing x.h
```

in place of a line that would normally show an include of this header.

If the program were to be linted subsequently with the same options, then instead of seeing a verbosity line indicating that **x.h** were included and **x.lph** were written we would see:

```
Absorbing file x.lph
```

reflective of the fact that **x.lph** contains binary information representative of the information in **x.h**.

7.1.4 The use of make files

The **.lph** file is not automatically regenerated when the original header (or any of its sub headers) is modified. If it is important that it must be done automatically then you will need a **make** facility or its equivalent. An entry in the **make** file could be as simple as:

```
x.lph:  x.h ...
        del x.lph
```

In words, an **x.lph** is composed of **x.h** plus any of its included header files and is 'manufactured' by a deletion of **x.lph**. If this confuses the **make** facility then you might try something like:

```
request.lph:  x.h ...
        del x.lph
        touch request.lph
```

Here you need to create a file called "**request.lph**" whose content is the minimal necessary for **make** to consider it a file. Whenever any of a collection of headers is modified, **x.lph** is deleted and the date of the **request.lph** is updated.

7.2 Bypass Headers

If all went well with a pre-compiled header, you needn't pursue the intricacies of bypass headers. If, for some reason, you decide that pre-compiled headers are not for you, either because they fail to operate properly or because you can't create a simple header file to represent all your standard includes or because your headers are always changing, your project may nonetheless be eligible for dramatic speed improvement by use of bypass headers.

A Bypass Header is a header that has been designated (by one of the three options listed below) to be a bypass header or is included by a bypass header. The purpose of bypass headers is to increase the speed of linting multiple module applications without compromising the quality of code analysis. Normal headers are physically scanned every time they are included (by the **#include** command). Bypass headers are physically scanned only in the first module in which they are included. In subsequent modules when a **#include** is seen for a bypass header, an attempt is made to recreate the effect of that header including all subheaders.

Since header processing, especially for large application frameworks, can represent the bulk of the time spent linting, the time savings can be quite substantial.

A header file can become a bypass header if:

(i) It falls within one of the four broad categories of the option **+bypclass**, viz. **all**, **ansi**, **angle** and **foreign** (described below), and is not excluded by either the **-bypdir** or the **-byph** option.

(ii) OR, for finer control, it comes from a directory specified with **+bypdir** and is not specifically excluded with **-byph**.

(iii) OR, for the finest control, it is specifically included by name via **+byph**.

(iv) OR, is included within a bypass header file.

The alert reader will note that this method of identifying files as bypass is very nearly identical to the way files are identified as library with the important difference that the prefix '**lib**' is replaced by the prefix '**byp**' throughout. Another important difference is that by default some headers (**angle** and **foreign**) are library whereas by default no headers are considered bypass.

You may determine whether header files are interpreted as bypass header files by using some variation of the **-vf** verbosity option. For each included bypass header you will receive a message similar to:

```
Including file c:\compiler\stdio.h (bypass)
```

The tag: '**(bypass)**' indicates a bypass header file. In subsequent modules, when the header is bypassed you will receive the message:

```
Bypassing file c:\compiler\stdio.h (bypass)
```

What follows is a more complete description of the three options used to specify if or when a header file is a bypass header file.

+bypclass(***identifier*** [**,** ***identifier***] ... **)** specifies the set or sets of header files that are assumed to be bypass header files. Each ***identifier*** can be one of:

ansi The 'standard' ANSI/ISO C header files, viz.

assert.h	**limits.h**	**stddef.h**
ctype.h	**locale.h**	**stdio.h**
errno.h	**math.h**	**stdlib.h**
float.h	**setjmp.h**	**string.h**
fstream.h	**signal.h**	**strstream.h**
iostream.h	**stdarg.h**	**time.h**

all All header files are regarded as being bypass headers.

angle All headers specified with angle brackets.

foreign All header files found in directories via the search .list (**-i** or **INCLUDE** as appropriate).

Thus, if the **#include** contains a complete path name then the header file is not considered '**foreign**'. To endow such a file with the bypass header property use the **+byph** option. For example, if you have

```
#include "\include\graph.h"
```

and you want this header to be regarded as a bypass header use the option:

```
+byph(\include\graph.h)
```

If a search list (specified with **-i** option or **INCLUDE**) is used to locate this file, it is considered foreign; otherwise it is not.

By default, **+bypclass()** is in effect. That is, no files are presumed to be bypassed.

+bypdir(***directory*** [**,** ***directory***] ... **)** activates
-bypdir(***directory*** [**,** ***directory***] ... **)** deactivate
the ***directory*** (or directories) specified. The notion of ***directory*** here is identical to that in the **-i** option. If a ***directory*** is activated, then all header files found within the directory will be regarded as bypass header files (unless specifically inhibited by the **-byph** option). It overrides the **+bypclass** option for that particular directory. For example:

```
+bypclass()
+bypdir( c:\compiler )
+byph( stdafx.h )
```

requests that no header files be regarded as bypass files except those coming from directory **c:\compiler** and the header **stdafx.h** (see **+byph** below).

The principles in operation here are similar to those that govern the options **libclass**, **libdir** and **libh**. See Section 6.1, "Library Header Files" in the manual, for more examples and a fuller explanation.

+byph(***file*** [**,** ***file***] ... **)** adds
-byph (*file*** [**,** ***file***] ... **)** removes
files from the set that would otherwise be determined from the **+bypclass** and **+/-bypdir** options. For example:

```
+bypclass( ansi, angle )
+byph( windows.h, graphics.h )
+byph( os.h )   -byph( float.h )
```

requests that the header files described as **ansi** or **angle** (except for **float.h**) and the individual header files: **windows.h**, **graphics.h** and **os.h** (even if not specified with angle brackets) will be taken to be bypass header files.

Wild card characters '*****' and '**?**' are supported.

The operation of **byph** is very similar to the operation of **libh**. See Section 6.1, "Library Header Files", for an extensive set of examples and a more detailed explanation of the determination of directory and name in the specification of these options.

7.2.1 Constraints on Bypass Headers

What this section will demonstrate is that there are several reasons why headers cannot arbitrarily be made bypassable. Our conclusion will be that the simplest policy that always works is to

(a) decide which headers are to be included in all modules,
(b) place includes of these headers (directly or indirectly) in a single header file, say **common.h**,
(c) place a **#include** of **common.h** at the beginning of every module.
(d) Use the option:

```
+byph(common.h)
```

as the only bypass option.

Certainly variations on this theme are possible but this scheme is a good starting point.

One of the constraints on Bypass Headers is that they declare or define the same entities in the same way each time they are used. For example, consider the header **a.h** defined below:

a.h:

```
#ifdef X
#define A X
#else
#define A 13
#endif
```

We have the potential for an inconsistent use of **a.h** but it's not clear that this alone would be a problem because the '**X**' in question might be defined on the command line. If the command line definition is the only definition of **X**, **a.h** is perfectly consistent. But suppose we have modules **x.c** and **y.c** as follows:

x.c:

```
#define X 27
#include "a.h"
...
```

y.c:

```
#include "a.h"
...
```

.
In this case, **a.h** should not be designated as bypass. If it were, then when **x.c** were processed we would conclude that **a.h** defined a macro A to be **27**. If later, **y.c** were encountered in the same linting, the bypassing of **a.h** would simply result in a 'reawakening' of macro A with its

original value (**27**). However, without bypassing, a separate scan of **a.h** would reveal that its value should be **13**.

If one follows the procedure prescribed in the opening of this section (the creation of a **common.h**) then this sort of thing can't happen, as the include of **common.h** is the first thing done in each module. If **a.h** is marked as bypass then it would be a part of **common.h** and so every instance of its use would be consistent, whether or not **X** was defined on the command line.

The above example contained an explicit **#define** prior to a header and so represents a fairly obvious case of a potential header inconsistency. The following contains a more subtle example. Suppose **u.c** and **v.c** are to be linted and contain the following text:

```
u.c:

   #include "c.h"
   #include "d.h"
   ...

v.c:

   #include "d.h"
   ...

c.h:

   #ifndef C_H
       #define C_H
       // defines many things.
       ...

d.h:

   #ifndef C_H
       #include "c.h"
   #endif
   // defines other things
   ...
```

When **u.c** is processed, **c.h** can define many things. It is protected from multiple inclusions by its own include guard, **C_H**. When **u.c** includes **"d.h"**, **d.h** defines only other things because it uses the include guard of **c.h** to avoid the repeated include of **c.h**. This is called an external include guard and is not the best of programming practices but is nevertheless done.

When **v.c** includes "**d.h**" and if **d.h** were a bypass header, it would have previously been recorded that **d.h** defined only 'other things'. The code within **v.c** will not have use of the many defines available within **c.h**.

Fortunately, this situation is diagnosed with Fatal Error **328** (Bypass header follows a different header sequence in two different modules). This message is fatal and discontinues processing. It is an abrupt reminder that an inconsistent header sequence that includes bypass headers can contain subtle problems that are not worth chasing down until a consistent header sequence is achieved.

Note, however, that if we employ a single common header (`common.h`) at the beginning of every module (as is advocated at the beginning of this section) this problem as well as others will be resolved.

8. LINT OBJECT MODULES

Please Note. Lint Object Modules can be used to improve the speed of processing previously linted modules. However their use has been largely supplemented by Pre-compiled headers (**7.1**) and /or Bypass Headers (**7.2**). These latter forms retain information required in inter function value-tracking whereas the lob does not. This system of Lint Object Modules is retained for backward compatibility.

8.1 What is a LOB?

A *Lint Object Module* is a summary (in binary form) of the external information within a C or C++ module (or modules). PC-lint/FlexeLint can then use this information to compare with other modules for consistency. For example, if module **alpha.c** consists of:

```
void beta(x)
    double x;
    {
    gamma(3);
    }
```

then the associated lint object module for **alpha.c** (call it **alpha.lob**) will contain information that **beta** was defined with a **double** argument returning **void** and **gamma** was called with an **int** constant argument. The object file will retain the name of the original module, line number information and the names of all included header files.

8.2 Why are LOB's used?

Lint Object Modules are used to speed up the processing of large multi-module programs. Consider Figure LOB-1, which shows a program consisting of 9 modules **a1.c** through **a9.c**. Rather than linting all the source modules together, the programmer has linted the modules separately producing a Lint Object Module for each source module. A typical command might be:

```
lint -u a1.c -oo
```

which produces **a1.lob**. The **-u** (unit checkout) option should always be used when producing a Lint Object Module. All the usual messages will be produced, appropriate to unit checkout. (You may need the option **-zero** or **-zero(500)** to insure producing the object module in spite of error messages.)

After the Lint Object Modules are produced, they must then be linted together to make sure they are all consistent with one another. This is also shown in the figure. This can be done with the command:

```
lint *.lob
```

This produces the inter-module messages. If a single change is made to any source module, say to **a1.c** then only one object module needs to be regenerated. This is then combined with all of the other Lint Object Modules. The time required to process the collection of Lint Object Modules is typically short, on the order of processing just one source module and so the time savings is substantial. The observant reader will note that this process lends itself to incremental linting through a make facility. This is discussed later.

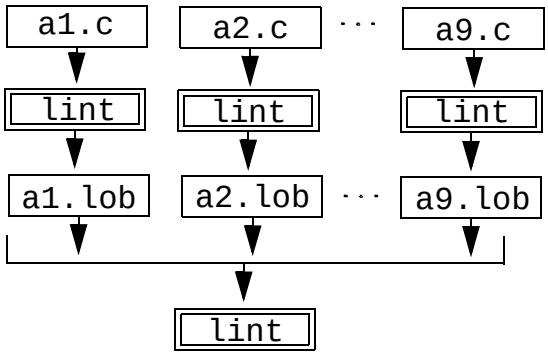


Figure **LOB-1**

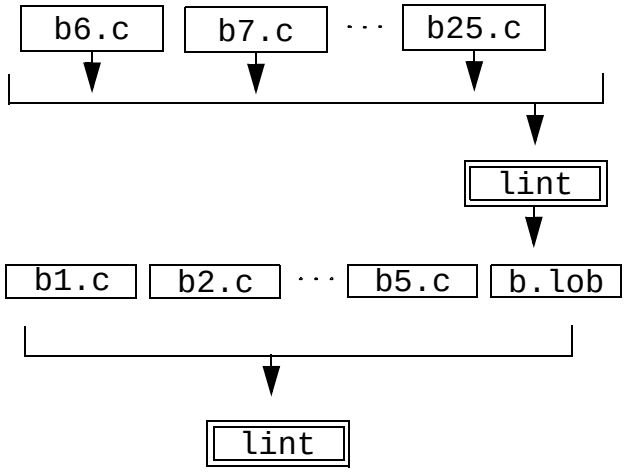


Figure **LOB-2**

Another way to use Lint Object Modules is shown in Figure LOB-2. Here a project consists of modules **b1.c** through **b25.c**. We assume our programmer is only responsible for modules **b1.c** through **b5.c** with other members of a team responsible for other modules. Accordingly a summary of the external information of modules **b6.c** through **b25.c** is captured in the Lint

Object Module **b.lob**. This is then used when linting **b1.c** through **b5.c** as is shown in the figure. This dramatically improves the speed of linting. It is instructive to compare this approach with that of producing function prototypes for all the functions in **b6.c** through **b25.c**. Function prototypes can be produced with the **-od** option (output declarations). Function prototypes do not contain information such as what line of what file contains information inconsistent with another file. It does not indicate which variables have been initialized or accessed or which objects have been referenced. The information in **.lob** files is, therefore, more complete and indicative than prototype information, as well as quite fast.

8.3 Producing a LOB

The option **-oo**[**(*filename*)**] will cause binary information for all modules on the command line to be output to the named file. The "**oo**" stands for "output object". If ***filename*** is omitted, as in the option:

```
-oo
```

then a name is formed from the first module name using an extension of "**.lob**". (a name ending in "**.lob**" is recommended since PC-lint/FlexeLint uses this extension on input to determine that the file is an object module and not a source module). For example:

```
lint -u alpha.c -oo
```

will output binary external information about **alpha.c** into the file **alpha.lob**.

8.4 -lobbase to reduce lob sizes

If there are many similar lob files, considerable space savings can be had by establishing a so-called lob base from which other files are derived. Consider the following example

```
c1.cpp:

  #include <gui.h>

  ...


c2.cpp:

  #include <gui.h>

  ...
```

Let us assume that **gui.h** expands to 30,000 lines of code (not untypical) and that **c1.cpp** and **c2.cpp** are more modest.

The command

```
lint -u c1.cpp -oo
```

produces a lint object module (**c1.lob**) of, say, 400,000 bytes. We could produce a second lob file **c2.lob** as

```
lint -u c2.c -oo
```

which would produce a second lob file of roughly equal volume. To save on storage space we can employ **c1.lob** as a base for the **c2.lob** using

```
lint -u -lobbase(c1.lob) c2.cpp -oo
```

The output in this case is "diff'ed" (transparently) against the base. The resulting **c2.lob** is considerably smaller (perhaps in the order of 10% of the original size). The only information carried in **c2.lob** is that which makes it different from **c1.lob**.

If the **-lobbase** option is given when a lob output is not requested it is ignored. This allows us, for example, to place the option

```
-lobbase(stdafx.lob)
```

within the library options file **lib-mfc.lnt** (Library options for Microsoft Foundation Classes) where it has no effect unless lob's are produced. (For those unfamiliar with MFC, **stdafx.cpp** is a short module whose only purpose is to include a range of header files).

What happens if the lob base is the same as the file we are creating?

For example

```
lint -u -lobbase(c1.lob) c1.cpp -oo
```

This would seem nonsensical as written but if the **-lobbase** option were part of a Library options file (as we have seen above in the case of **lib-mfc.lnt**) then it could easily occur. If the first name of the lob base ('**c1**' in this example) matches the first name of the lob that is produced, then a difference file is not emitted. Rather, the full and complete lob file is produced.

A difference file such as **c2.lob** contains the name of the lob base and also a hash number identifying the state of the base. If the base changes then presumably the hash number changes and PC-lint/FlexeLint will refuse to process any lob file based on the prior hash. The user needs to produce new lob files for each file dependent upon that base. See Section 8.5 Make Files to see how these dependencies can be automated.

8.5 Make Files

Lint Object Modules are well adapted for use with a make facility. For example, a ***make*** script, which follows the Unix ***make*** conventions can be of the following form. (Note: if you are following the Microsoft ***make*** conventions place the directive starting with **"project.lob"** at the end of the script).

```
.c.lob:
     lint -u make.lnt $* -oo

project.lob: module1.lob module2.lob module3.lob
     lint make.lnt *.lob

module1.lob: module1.c

module2.lob: module2.c

module3.lob: module3.c
```

where **make.lnt** contains

```
 -os(temp) +vm std.lnt
```

Here a program consists of three modules: **module1.c**, **module2.c** and **module3.c**. If any of these modules is altered, the lint command on the 2nd line is executed. If no flaws are found, the option **-oo** will cause object module **module*i*.lob** to be written.

The second request within the make file seems to suggest that **project.lob** is created during that step. **project.lob** is a fictitious name, which forces this command to take place. We could have produced a **project.lob** with **-oo(project.lob)** but it wouldn't be particularly useful.

The file **make.lnt** houses lint options used for linting within a make file. The **-os(*filename*)** has the effect of redirecting the messages (much as **>*filename***). Unlike redirection, the option can be placed within an indirect file as shown here. The file **std.lnt** contains standard lint options for this project in a manner that is discussed in Chapter 16. LIVING WITH LINT.

If you are using a lob base you should make other lob files dependent on the lob base file. The following example shows the make script above modified such that **module1.lob** is the lob base for module 2 and module 3.

```
.c.lob:
     lint -u make.lnt $* -oo
```

```
project.lob: module1.lob module2.lob module3.lob
     lint make.lnt *.lob

module1.lob: module1.c

module2.lob: module2.c module1.lob

module3.lob: module3.c module1.lob
```

where **make.lnt** contains

```
-os(temp) +vm std.lnt -lobbase(module1.lob)
```

A Lint Object Module will normally not be produced if as much as one lint error message is produced. If you want the ***make*** script to go on in spite of some messages, then you may use the option: **-zero** to force an exit code of zero. You probably don't want to use the **-os()** option in this case since your messages will be overwritten by the next command. Use **+os(temp)** (append to file **temp**) on the command line instead. Alternatively, you may want to parameterize the **-zero** option. An option of **-zero(*n*)**, as in **-zero(700)**, will have the effect of not counting messages whose message number is equal to or higher than the specified number ***n***, 700 in this example. This is much like a compiler producing warnings but going on to produce an object module as well.

8.6 Library Modules

Library modules are used to describe libraries and are usually for K&R compilers. They contain the option **/*lint -library*/** or its equivalent. These can also be in object form. For compilers that support prototypes, library modules are becoming obsolete because a header file (or files) describing the library is generally all that lint needs to determine whether function calls are compatible with a library. For compilers that do not support prototypes, it is necessary to have an extra module that describes arguments to library functions. Call this **sl.c**. For large library modules it makes sense to produce an object version of this module. The command

```
lint co.lnt sl.c -oo
```

(where **co.lnt** is a compiler options file) will produce the file **sl.lob**. Declarations of objects that are not referenced are not normally a part of a Lint Object Module. But, in this case, where the only input modules are library modules, an exception to this rule is made. For example suppose **sl.c** contains:

```
/*lint -library */

double sin(double);
```

Since **sin** is neither defined nor referenced within the module (it is only declared) it would not normally be retained in an object module. But because this is a library module and because there are no non-library modules being presented to PC-lint/FlexeLint, all library declarations are retained.

8.7 Options for LOB's

To conserve on space, Lint Object Modules do not, by default, contain objects that have merely been declared (but not referenced or defined). The option **+fod** (Object module receives all Declarations) overrides this default behavior. Also, by default, library objects (unless referenced or defined) are not normally included, again to save space. **+fol** forces all library symbols to be included in the module. This option is not normally needed because when making an object module from only library modules, the flag is automatically thrown on. See Section 6.3 Library Object Modules

8.8 Limitations of LOB's

To conserve on space, macros, typedefs and templates are not placed within Lint Object Modules. This affects some Informational messages (**755** global macro not referenced, **767** macro was defined differently in another module and others). For this reason you will occasionally want to lint all your source files together even though your normal modus operandi is to use Lint Object Modules.

9. STRONG TYPES

Strong type checking is gold
Normal type checking is silver
But casting is brass

9.1 Quick Start

If you want a quick action-packed introduction to strong types use the option:

```
-strong(AJX,typename)
```

where ***typename*** is one of your **typedef** types and observe the messages produced. This option should be positioned ***before*** the first source file.

Then read on to find out what this means and how to control the messages.

Sad Note: If you are using a batch file to invoke PC-lint such as **lin.bat** you will not get the comma past the command interpreter. Instead use '**!**' as in

```
-strong(AJX!typename)
```

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

9.3 -strong

-strong(***flags*** [**,** ***name***] … **)**

identifies each ***name*** as a strong type with properties specified by ***flags***. Presumably there is a later **typedef** defining any such ***name*** to be a type. This option has no effect on **typedef**'s defined earlier. If ***name*** is omitted, then ***flags*** specifies properties for all **typedef** types that are not identified by some other **-strong** option. Please note, this option must come ***before*** the **typedef**.

The ***flags*** can be:

A Issue a warning upon some kind of Assignment to the strong type. (includes assignment operator, return value, argument passing, initialization). **A** may be followed by one or more of the following letters, which soften the meaning of **A**.

- **i** ignore Initialization.
- **r** ignore Return statements.
- **p** ignore argument Passing.
- **a** ignore the Assignment operator.
- **c** ignore assignment of Constants. (constants include integral constants, quoted strings and expressions of the form: **&x** where **x** is a static or automatic variable.)
- **z** ignore assignment of Zero. A zero is defined as any zero constant that has not been cast to a strong type. For example, **0L** and **(int) 0** are considered zero but **(HANDLE) 0** where **HANDLE** is a strong type is not. Also, **(HANDLE*) 0** is not considered zero.

As an example, **-strong(Ai,BITS)** will issue a warning whenever a value whose type is not **BITS** is assigned to a variable whose type is **BITS** except when initialized.

X Check for strong typing when a value is eXtracted. This causes a warning to be issued when a strongly typed value is assigned to a variable of some other type (in one of the four ways described above). But note, the softeners (**i, r, p, a, c, z**) cannot be used with **X**.

J Check for strong typing when a value is Joined (i.e., combined) with another type across a binary operator. This can be softened with one or more of the following lower-case letters immediately following the **J**:

- **e** ignore Equality operators (**==** and **!=**) and the conditional operator (**?:**).

r ignore the four Relational operators (**> >= < <=**).

m ignore the three multiplication operators (*** / %**)

d The strong type will be considered a dimension for the purpose of dimensional analysis (See Section 9.4.4 Dimensional Analysis). This is the normal default

n The strong type will be considered dimensionally neutral. See Section 9.4.2 Dimensionally Neutral (Jn).

a The strong type will be considered anti-dimensional (See Section 9.4.3 Anti-Dimensional (Ja). This would be the default if option **-fdd** were given turning off the Dimension by Default flag.

o ignore the Other binary operators, which are the two additive operators (**+ -**) and the three bit-wise operators (**| & ^**).

c ignore combining with Constants.

z ignore when combining with a Zero value. See the 'A' flag above for what constitutes a zero.

The meaning of **Jo** has changed in the transition from Version 8 to Version 9. In Version 8 **Jo** controlled the three multiplication operators together with the other operators that it now controls. In Version 9 the multiplication operators, as indicated above, are by default under the control of **Jm**. To obtain the old behavior use the option **-fjm**.

B The type is Boolean. Normally only one ***name*** would be provided and normally this would be used in conjunction with other flags. (If through the fortunes of using a third party library, multiple Boolean's are thrust upon you, make sure these are related through a type hierarchy. See Section 9.6 Type Hierarchies) The letter '**B**' has two effects:

1. Every Boolean operator will be assumed, for the purpose of strong type-checking, to return a type compatible with this type. The Boolean operators are those that indicate true or false and include the four Relational and two Equality operators mentioned above, Unary **!**, and Binary **&&** and **||**.

2. Every context expecting a Boolean, such as an **if** clause, **while** clause, second expression of a **for** statement, operands of Unary **!** and Binary **||** and **&&**, will expect to see this strong type or a warning will be issued.

b This is like flag **B** except that it has only effect number 1 above. It does not have effect 2. Boolean contexts do not require the type.

Flag **B** is quite restrictive insisting as it does that all Boolean contexts require the indicated Boolean type. By contrast, flag **b** is quite permissive. It insists on nothing by itself and serves to identify certain operators as returning a type strongly compatible with the strong type. See also the '**l**' flag below.

l is the Library flag. This designates that the objects of the type may be assigned values from or combined with library functions (or objects) or may be passed as arguments to library functions. The usual scenario is that a library function is prototyped without strong types and the user is passing in strongly typed arguments. Presumably the user has no

control over the declarations within a library. Also, this flag is necessary to get built-in predicates such as **isupper** to be accepted with flag **B**. See the example below. See also Section 6.1 Library Header Files for a definition of library.

f goes with **B** or **b** and means that bit fields of length one should not be Boolean (otherwise they are). See Bit field example below.

These flags may appear in any order except that softeners for **A** and **J** must immediately follow the letter. There is at most one '**B**' or '**b**'. If there is an '**f**' there should also be a '**B**' or '**b**'. In general, lower-case letters reduce or soften the strictness of the type checking whereas upper-case letters add to it. The only exceptions are possibly '**b**' and '**f**' where it is not clear whether they add or subtract strictness.

If no flags are provided, the type becomes a 'strong type' but engenders no specific checking other than for declarations.

Examples of -strong

For example, the option

```
-strong(A)
```

indicates that, by default, all **typedef** types are checked on Assignment (**A**) to see that the value assigned has the same **typedef** type.

The options:

```
-strong(A)  -strong(Ac,Count)
```

specify that all **typedef** types will be checked on Assignment and constants will be allowed to be assigned to variables of type **Count**.

As another example,

```
-strong(A)  -strong(,Count)
```

removes strong checking for **Count** but leaves Assignment checking on for everything else. The order of the options may be inverted. Thus

```
-strong(,Count)  -strong(A)
```

is the same as above.

Consider:

```
//lint -strong(Ab,Bool)
typedef int Bool;

Bool gt(a,b)
    int a, b;
    {
    if(a) return a > b;  // OK
    else return 0;       // Warning
    }
```

This identifies **Bool** as a strong type. If the flag **b** were not provided in the **-strong** option, the result of the comparison operator in the first **return** statement would not have been regarded as matching up with the type of the function. The second **return** results in a Warning because **0** is not a **Bool** type. An option of **-strong(Acb,Bool)**, i.e. adding the **c** flag, would suppress this warning.

We do not recommend the option '**c**' with a Boolean type. It is better to define

```
#define False (bool) 0
```

and

```
return False;
```

Had we used an upper-case **B** rather than lower-case **b** as in:

```
-strong( AB, Bool )
```

then this would have resulted in a Warning that the **if** clause **if(a)**... is not Boolean (variable **a** is **int**). Presumably we should write:

```
if( a != 0 ) ...
```

As another example:

```
/*lint -strong( AJXl, STRING ) */
typedef char *STRING;
STRING s;

    ...
    s = malloc(20);
    strcpy( s, "abc" );
```

Since **malloc** and **strcpy** are library routines, we would ordinarily obtain strong type violations when assigning the value returned by **malloc** to a strongly typed variable **s** or when passing the strongly typed **s** into **strcpy**. However, the **l** flag suppresses these strong type clashes.

Strong types can be used with bit fields. Bit fields of length one are assumed to be, for the purpose of strong type checking, the prevailing Boolean type if any. If there is no prevailing Boolean type or if the length is other than one, then, for the purpose of strong type checking, the type is the bulk type from which the fields are carved. Thus:

```
//lint -strong( AJXb, Bool )
//lint -strong( AJX, BitField )

typedef int Bool;
typedef unsigned BitField;

struct foo
    {
    unsigned a:1, b:2;
    BitField c:1, d:2, e:3;
    } x;

void f()
    {
    x.a = (Bool) 1;     // OK
    x.b = (Bool) 0;     // strong type violation
    x.a = 0;            // strong type violation
    x.b = 2;            // OK
    x.c = x.a;          // OK
    x.e = 1;            // strong type violation
    x.e = x.d;          // OK
    }
```

In the above, members **a** and **c** are strongly typed **Bool**, members **d** and **e** are typed **BitField** and member **b** is not strongly typed.

To suppress the Boolean assumption for one-bit bit fields use the flag '**f**' in the **-strong** option for the Boolean. In the example above, this would be **-strong(AJXbf,Bool)**.

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.

9.4.2 Dimensionally Neutral (Jn)

A ***dimensionally neutral*** type is a strong type such that when multiplied or divided by a dimension will act as a non-strong type.

For example:

```
//lint -strong( AJdX, Sec )
typedef double Sec;

//lint -strong( AJnX, Cycles )
typedef double Cycles;
Cycles n;
Sec t;
...
t = n * t;          // OK, Cycles are neutral
t = t / n;          // still OK.
n = n / t;          // warning: '1/Sec' assigned to 'Cycles'
```

The `n` softener of the `J` flag as in the `AJnX` sequence above designates that type `Cycles` is dimensionally neutral and will drop away when combined multiplicably with the dimension

Cycles as shown in the first two assignments. However, **Cycles** acts as a strong type in every other regard. An illustration of this is the last line in this example which produces a warning that the type **'1/Sec'** is being assigned to **Cycles**.

Thus, **Cycles** is playing the role that it traditionally plays in Physics and Engineering. It contains no physical units and when multiplied or divided by a dimension does not change the dimensionality of the result.

9.4.3 Anti-Dimensional (Ja)

An ***anti-dimensional*** type is a strong type that when multiplied or divided is expected to be combined with the same type, or one that is compatible through the usual strong type hierarchies. It functions in this regard much like addition and subtraction.

For example:

```
//lint -strong( AJaX, Integer )
typedef int Integer;
Integer k;
int n;
...
k = k * k;      // OK
k = n * k;      // warning: Integer joined with non-Integer
```

The sequence **Ja** in the above indicates that **Integer** is anti-dimensional. This form of behavior is identical to that of PC-lint /FlexeLint Version 8 and earlier. To obtain anti-dimensionality by default you may use the option:

```
-fdd        // turn off the Dimension by Default flag
```

If you have many strong types in your application from some version prior to Version 9, you will probably want to use this option to obtain the previous interpretation by default. Then, one by one, you can change individual types to Dimension (**Jd** or Dimensionally Neutral (**Jn**) as time and inclination dictate.

9.4.4 Dimensional Analysis

The strong type mechanism can support the traditional dimensional analysis exploited by physicists, chemists and engineers. When strong types are added, subtracted, compared or assigned, the strong types need merely match up with each other. However, multiplication and division can join arbitrary dimensional types and the result is often a new type. Consider forming the velocity from a distance and a time:

```
//lint -strong( AcJcX, Met, Sec, Velocity = Met/Sec )
typedef double Met, Sec, Velocity;
Velocity speed( Met d, Sec t )
    {
    Velocity v;
    v = d / t;              // ok
    v = 1 / t;              // warning
    v = (3.5/t) * d;        // ok
    v = (1/(t*t)) * d * t;  // ok
    return v;               // ok
    }
```

In this example, the 4th argument to the **-strong** option:

```
Velocity = Met/Sec
```

relates strong type **Velocity** to strong types **Met** and **Sec**. This particular suboption actually creates two strong types: **Velocity** and **Met/Sec** and relates the two types by making **Met/Sec** the parent type of **Velocity**. This relationship can be seen in the output obtained from the option **-vh** (or the compact form **-vh-**). As an example the results of the **-vh** option for the above example are: (Non-graphic characters were used in the output using the option **-fhg**)

```
-  Met

-  Sec

-  Met/Sec
   |
   + - Velocity

-  1/Sec

-  (Sec*Sec)

-  1/(Sec*Sec)

-  Met/(Sec*Sec)
```

The division of **Met** by **Sec** (within the option) can be produced in many equivalent ways. E.g.

```
Velocity = (1/Sec) * Met
Velocity = ((1/Sec) * (Met))
Velocity = (Met/(Sec*Sec)) * Sec
```

are all equivalent. All of these dimensional expressions are reduced to the canonical form **Met/Sec** which was the form given in the original option. Note that parentheses can be used freely and in some cases must be used to obtain the correct results. E.g.

```
Acceleration = Met/Sec*Sec          // wrong
Acceleration = Met/(Sec*Sec)        // correct
```

We follow C syntactic rules where the operators bind left to right and the example labeled 'wrong' results, after cancellation, in just **Met**.

Briefly and for the record the canonical form produced is:

(F1*F2*...*F*n*)/(G1*G2*...*G*m*)

where each **F*i*** and each **G*i*** are simple single-identifier sorted strong types and where ***n*** >= 0 and ***m*** >= 0 but if ***n*** is less than 2 the upper parentheses are dropped out and if ***m*** is less that 2 the lower parentheses are dropped and if ***n*** is 0 the numerator is reduced to **1** and if ***m*** is 0 the entire denominator including the **/** is dropped.

Returning to our original example (the function **speed**), when the statement:

```
v = d/t;
```

is encountered and an attempt is made to evaluate **d/t** the dimensional nature of the types of the two arguments is noted and the names of these types is combined by the division operator to produce **"Met/Sec"**. This uses essentially the same algorithms and canonicalization as the compound type analysis with a **-strong** option. The resulting type is assigned to **Velocity** without complaint because of the previously described parental relationship that exists between these two strong types.

In the next statement

```
v = (3.5/t) * d;
```

the division results in the creation of a new strong type (**1/Sec**) which when multiplied by **Met** will become **Met/Sec**. The created type will have properties **AJcdX** and the underlying type will be the type that a compiler would compute.

9.4.5 Conversions

A simple example in the use of Dimensional strong types is providing a fail-safe method of converting from one system of units to another. Such conversions can quite often be accomplished by a single numeric factor. Such conversion factors should have dimensions attached to prevent mistakes. E.g.

```
//  Centimeters to/from Inches
//lint -strong( AJdX, In, Cm, CmPerIn = Cm/In )
typedef double In, Cm, CmPerIn;
CmPerIn cpi = (CmPerIn) 2.54;     // conversion factor
void demo( In in, Cm cm )
```

```
{
...
in = cm / cpi;              // convert cm to in
...
cm = in * cpi;              // convert in to cm
...
}
```

In this example we are defining a conversion factor, **cpi**, that will allow us to convert inches to centimeters (by multiplication) and convert centimeters to inches (via division). Without strong types, conversion factors can be misused. Do I multiply or divide? Using strong types you can be assured of getting it right.

Obviously not all conversions fall into the category of being described by a conversion factor. Conversions between Celsius and Fahrenheit, for example, require an expression and this typically means defining a pair of functions as in the following:

```
//lint -strong( AJdX, Fahr, Celsius )
typedef double Fahr, Celsius;
Celsius toCelsius( Fahr t )
    { return (t-(Fahr)32.) * (Celsius)5. / (Fahr)9.; }
Fahr toFahr( Celsius t )
    { return (Fahr)32. + t * (Fahr)9. / (Celsius)5.; }
```

The function call overhead is probably not significant, but if it is, you may declare the functions to be inline in C++. Some C systems support inline functions, but in any case, you can use macros.

9.4.6 Integers

Although the examples of dimensional analysis offered above refer to floating point quantities, the same principles apply to integer arithmetic. E.g.

```
#include <stdio.h>
#include <limits.h>

//lint -strong( AcJdX, Bytes, Bits )
//lint -strong( AcJdX, BitsPerByte = Bits / Bytes )
typedef size_t Bytes, Bits, BitsPerByte;
BitsPerByte bits_per_byte = CHAR_BIT;
Bytes size_int = sizeof(int);
Bits length_int = size_int * bits_per_byte;
```

In this example **Bits** is the length of an object in bits and **Bytes** is the length of an object in bytes. **bits_per_byte** becomes a conversion factor to translate from one unit to the other. The example shows the use of that conversion factor to compute the number of bits in an integer.

Let's say that you wanted to strengthen the integrity and robustness of a program by making sure that all shifts were by quantities that were typed **`Bits`**. For example you could define a function **`shift_left`** with the intention that this function have a monopoly on shifting **`unsigned`** types to the left. This could take the form:

```
inline unsigned shift_left( unsigned u, Bits b )
    {
    return u << b;
    }
```

A simple grep for **`"<<"`** can be used to ensure that no other shift lefts exist in your program. Note that the example deals only with **`unsigned`** but if there were other types that you wanted to shift left, such as **`unsigned long`**, you can use the C++ overload facility.

Using C you may also employ the **`shift_left`** function. However you may not have **`inline`** available and you may be concerned about speed. To obtain the required speed you can employ a macro as in:

```
#define Shift_Left(u,b) ((u) << (b))
```

But you will note that there is now no checking to ensure that the number of bits shifted are of the proper type. One approach is to use conditional compilation:

```
#ifdef _lint
#define Shift_Left(u,b) shift_left(u,b)
#else
#define Shift_Left(u,b) ((u) << (b))
#endif
```

This will work adequately in C. If the quantity being shifted is anything other than plain **`unsigned`**, you will need to duplicate this pattern for each type.

A probably better approach is to define a macro that can check the type, such as the macro **`Compatible`** defined below:

```
#ifdef _lint
  #define Compatible(e,type) (*(type*)__Compatible = (e),(e))
  static char __Compatible[100];
  //lint -esym(528,__Compatible)  don't complain if not referenced
  //lint -esym(551,__Compatible)  don't complain if not accessed
  //lint -esym(843,__Compatible)  could be const?
#else
  #define Compatible(e,type) (e)
#endif
```

You could then define the original **`Shift_Left`** macro as:

```
#define Shift_Left(u,b) ((u) << Compatible(b,Bits))
```

`Compatible(e,type)` works as follows. Under normal circumstances (i.e. when compiling) it is equivalent to the expression **`e`**. When linting it is also equivalent to **`e`** except that there is a side effect of assigning to some obscure array that has been artfully configured into resembling a data object of type **`type`**. A complaint will be issued if the expression **`e`** would draw a complaint when assigned to an object of type **`type`**.

In this way you can be assured that the shift amount is always assignment compatible with **`Bits`**. Note that there is no longer a need for the twin **`Shift_Left`** definitions. And **`Compatible`** can be used in many other places to assure that objects are typed according to program requirements.

For simplicity, we have focused on shifting left. Obviously, similar comments can be made for shifting right.

9.4.7 Migrating to Dimensions

Users of strong types may experience "Version Shock" when encountering Version 9 from some prior version. This is probably a result of strong types being multiplied or divided. In earlier versions dimensional analysis was not supported and all strong types were implemented in a fashion that we now call anti-dimensional. In Version 9 these types are taken as dimensional by default. The best temporary approach is to use the option:

```
-fdd   // not strong Dimension by Default
```

This will restore the default to the prior state; i.e. all strong types not explicitly indicated will be considered anti-dimensional.

But ultimately you will want to eliminate this crutch. This requires separating out two distinct uses of strong typing. One use wants nothing more than to ensure that computations are performed using a consistent underlying type. For this usage it is perfectly OK to multiply the two identical strong types together and obtain as a result the same strong type. For example, consider a type **`Int32`** that represents integral quantities of length 32 bits. In such a case you should denote the type as antidimensional using the coding **`'Ja'`** as in the following example:

```
//lint -strong( AJaX, Int32 )
Int32 x;
Int32 y;
...
x = x * y;      // no complaints
```

But other uses are probably dimensional or dimensionally neutral. Consider the aforementioned **`Bytes`** type that represents the length of an object in bytes. You may find complaints about code which in fragment form resembles:

```
Count count;        // a number of items
```

```
Bytes sz1;          // length of a single item
Bytes sz;           // length of all the items
...
sz = (Bytes) count * sz1;  // reluctant cast
```

The assignment in the statement 'reluctant cast' now draws complaints that a **(Bytes*Bytes)** is being assigned to a **Bytes**. The cast is denoted 'reluctant' because it was originally used only to fulfill the earlier requirement that both sides of the multiplication operator needed the same strong type.

So the natural solution is to designate **Bytes** to be a dimension with the **Count** type dimensionally neutral. This would require options similar to the following:

```
//lint -strong( AJdX, Bytes )   Bytes is dimensional
//lint -strong( AJnX, Count )   Count is dimensionally neutral
```

9.5 -index

Description

-index(***flags, ixtype, sitype*** [**,** ***sitype***] … **)**

This option is supplementary to and can be used in conjunction with the **-strong** option. It specifies that ***ixtype*** is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type ***sitype*** (or ***sitype***'s if more than one is provided). Both the ***ixtype*** and the ***sitype*** are assumed to be names of types subsequently defined by a **typedef** declaration. ***flags*** can be

c allow Constants as well as ***ixtype***, to be used as indices.

d allow array Dimensions to be specified without using an ***ixtype***.

Examples of -index

For example:

```
//lint -strong( AzJcX, Count, Temperature )
//lint -index( d, Count, Temperature )
//        Only Count can index a Temperature

typedef float Temperature;
typedef int Count;
Temperature t[100];              // OK because of d flag
Temperature *pt = t;             // pointers are also checked
```

```
                                        // ... within a function
    Count i;

    t[0] = t[1];                        // Warnings, no c flag
    for( i = 0; i < 100; i++ )
    t[i] = 0.0;                         // OK, i is a Count
    pt[1] = 2.0;                        // Warning
    i = pt - t;                         // OK, pt-t is a Count
```

In the above, **Temperature** is said to be ***strongly indexed*** and **Count** is said to be a ***strong index***.

If the **d** flag were not provided, then the array dimension should be cast to the proper type as for example:

```
Temperature t[ (Count) 100 ];
```

However, this is a little cumbersome. It is better to define the array dimension in terms of a manifest constant, as in:

```
#define MAX_T  (Count) 100
Temperature t[MAX_T];
```

This has the advantage that the same **MAX_T** can be used in the **for** statement to govern the range of the **for**.

Note that pointers to the Strongly Indexed type (such as **pt** above) are also checked when used in array notation. Indeed, whenever a value is added to a pointer that is pointing to a strongly indexed type, the value added is checked to make sure that it has the proper strong index.

Moreover, when strongly indexed pointers are subtracted, the resulting type is considered to be the common Strong Index. Thus, in the example,

```
i = pt - t;
```

no warning resulted.

It is common to have parallel arrays (arrays with identical dimensions but different types) processed with similar indices. The **-index** option is set up to conveniently support this. For example, if **Pressure** and **Voltage** were types of arrays similar to the array **t** of **Temperature** one might write:

```
//lint -index( , Count, Temperature, Pressure, Voltage )
...
```

```
Temperature t[MAX_T];
Pressure p[MAX_T];
Voltage v[MAX_T];
...
```

Multidimensional Arrays

The indices into multidimensional arrays can also be checked. Just make sure the intermediate type is an explicit **typedef** type. An example is **Row** in the code below:

```
/*  Types to define and access a 25x80 Screen.
a Screen is 25 Row's
a Row is 80 Att_Char's */

/*lint -index( d, Row_Ix, Row )
-index( d, Col_Ix, Att_Char ) */

typedef unsigned short Att_Char;
typedef Att_Char Row[80];
typedef Row Screen[25];
typedef int Row_Ix;            /* Row Index */
typedef int Col_Ix;            /* Column Index */

#define BLANK (Att_Char) (0x700 + ' ')

Screen scr;
Row_Ix row;
Col_Ix col;

void main()
{
int i = 0;

scr[ row ][ col ] = BLANK;     /* OK */
scr[ i ][ col ] = BLANK;       /* Warning */
scr[col][row] = BLANK;         /* Two Warnings */
}
```

In the above, we have defined a **Screen** to be an array of **Row**'s. Using an intermediate type does ***not*** change the configuration of the array in memory. Other than for type-checking, it is the same as if we had written:

```
typedef Att_Char Screen[25][80];
```

9.6 Type Hierarchies

9.6.1 The Need for a Type Hierarchy

Consider a ***Flags*** type, which supports the setting and testing of individual bits within a word. An application might need several different such types. For example, one might write:

```
typedef unsigned Flags1;
typedef unsigned Flags2;
typedef unsigned Flags3;

#define A_FLAG (Flags1) 1
#define B_FLAG (Flags2) 1
#define C_FLAG (Flags3) 1
```

Then, with strong typing, an **A_FLAG** can be used with only a **Flags1** type, a **B_FLAG** can be used with only a **Flags2** type, and a **C_FLAG** can be used with only a **Flags3** type. This, of course, is just an example. Normally there would be many more constants of each ***Flags*** type.

What frequently happens, however, is that some generic routines exist to deal with ***Flags*** in general. For example, you may have a stack facility that will contain routines to push and pop ***Flags***. You might have a routine to print ***Flags*** (given some table that is provided as an argument to give string descriptions of individual bits).

Although you could cast the ***Flags*** types to and from another more generic type, the practice is not to be recommended, except as a last resort. Not only is a cast unsightly, it is hazardous since it suspends type-checking completely.

9.6.2 The Natural Type Hierarchy

The solution is to use a type hierarchy. Define a generic type called **Flags** and define all the other ***Flags*** in terms of it:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef Flags Flags3;
```

In this case **Flags1** can be combined freely with **Flags**, but not with **Flags2** or with **Flags3**.

Hierarchy depends on the state of the **fhs** (Hierarchy of Strong types) flag, which is normally ON. If you turn it off with the

```
-fhs
```

option the natural hierarchy is not formed.

We say that **Flags** is a ***parent*** type to each of **Flags1**, **Flags2** and **Flags3**, which are its ***children***. Being a parent to a child type is similar to being a base type to a derived type in an object-oriented system with one difference. A parent is normally interchangeable with each of its children; a parent can be assigned to a child and a child can be assigned to a parent. But a base type cannot normally be assigned to a derived type. But even this property can be obtained via the **-father** option (See Section 9.6.4 Restricting Down Assignments (-father)).

A generic ***Flags*** type can be useful for all sorts of things, such as a generic zero value, as the following example shows:

```
//lint -strong(AJX)
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
#define FZERO (Flags) 0
#define F_ONE (Flags) 1

void m()
{
Flags1 f1 = FZERO;          //  OK
Flags2 f2;

f2 = f1;                    // Warn
if(f1 & f2)                 // Warn because of J flag
f2 = f2 | F_ONE;            //  OK
f2 = F_ONE | f2;            //  OK Flag2 = Flag2
f2 = F_ONE | f1;            // Warn Flag2 = Flag1
}
```

Note that the type of a binary operator is the type of the most restrictive type of the type hierarchy (i.e., the child rather than the parent). Thus, in the last example above, when a **Flags** OR's with a **Flags1** the result is a **Flags1**, which clashes with the **Flags2**.

Type hierarchies can be an arbitrary number of levels deep.

There is evidence that type hierarchies are being built by programmers even in the absence of strong type-checking. For example, the header for Microsoft's Windows SDK, **windows.h**, contains:

```
...
```

```
typedef unsigned int        WORD;
typedef WORD                ATOM;
typedef WORD                HANDLE;
typedef HANDLE              HWND;
typedef HANDLE              GLOBALHANDLE;
typedef HANDLE              LOCALHANDLE;
typedef HANDLE              HSTR;
typedef HANDLE              HICON;
typedef HANDLE              HDC;
typedef HANDLE              HMENU;
typedef HANDLE              HPEN;
typedef HANDLE              HFONT;
typedef HANDLE              HBRUSH;
typedef HANDLE              HBITMAP;
typedef HANDLE              HCURSOR;
typedef HANDLE              HRGN;
typedef HANDLE              HPALETTE;
...
```

9.6.3 Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via **`typedef`** declaration has a limitation. All the types in a single tree must be the same underlying type. The **`-parent`** option can be used to supplement (or completely replace) the strong type hierarchy established via **`typedef`** declarations.

An option of the form:

`-parent(` ***`Parent, Child`*** [**`,`** ***`Child`***] … **`)`**

where ***`Parent`*** and ***`Child`*** are type names defined via **`typedef`** will create a link in the strong type hierarchy between the ***`Parent`*** and each of the ***`Child`*** types. The ***`Parent`*** is considered to be equivalent to each ***`Child`*** for the purpose of Strong type matching. The types need not be the same underlying type and normal checking between the types is unchanged.

A link that would form a loop in the tree is not permitted.

For example, given the options:

```
-parent(Flags1,Small)
-strong(AJX)
```

and the following code:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef unsigned char Small;
```

then the following type hierarchy is established:

```
      Flags
      /   \
Flags1   Flags2
  |
Small
```

If an object of type **Small** is assigned to a variable of type **Flags1** or **Flags**, no strong type violation will be reported. Conversely, if an object of type **Flags** or **Flags1** is assigned to type **Small**, no strong type violation will be reported but a loss of precision message will still be issued (unless otherwise inhibited) because normal type checking is not suspended.

If the **-fhs** option is set (turning off the hierarchy of strong types flag) a **typedef** will not add a hierarchical links. The only links that will be formed will be via the **-parent** option.

9.6.4 Restricting Down Assignments (-father)

The option

-father(***Parent, Child*** [**,** ***Child***] … **)**

is similar to the **-parent** option and has all the effects of the **-parent** option and has the additional property of making each of the links from ***Child*** to ***Parent*** one-way. That is, assignment from ***Parent*** to ***Child*** triggers a warning. You may think of **-father** as a strict version of **-parent**.

The rationale for this option is shown in the following example.

```
typedef int FIndex;
typedef FIndex Index;
```

Here **Index** is a special Index into an array. **FIndex** is a Flag or an **Index**. If negative, **FIndex** is taken to be a special flag and otherwise can take on any of the values of **Index**. By defining **Index** in terms of **FIndex** we are implying that **FIndex** is the parent of **Index**. The reader not accustomed to OOP may think that we have the derivation backwards, that the simpler **typedef**, **Index**, should be the parent. But **Index** is the more specific type; every **Index** is an **FIndex** but not conversely. Whereas it is expected that we can assign from **Index** to **FIndex** it could be dangerous to do the inverse.

Since we don't want down assignments we give the option

```
-father( FIndex, Index )
```

in addition to the strong options, say

```
-strong( AcJcX, FIndex, Index )
```

Then

```
FIndex n = -1;
Index  i =  3;

i = n;          /* Warning */
n = i;          /* OK */
```

The safe way to convert a **FIndex** to **Index** is via a function call as in

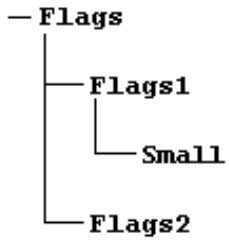
```
Index  F_to_I( FIndex fi )
           { return (Index)(fi >= 0 ? fi : 0); }
```

Then, although we need to use a cast in this function we need not use a cast in the rest of the program.

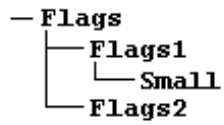
The net result of all this is that although flags and indices occupy the same storage location, we will never use a flag where an index is needed.

9.6.5 Printing the Hierarchy Tree

To obtain a visual picture of the hierarchy tree, use the letter **'h'** in connection with the **-v** option. For example, using the option **+vhm** for the example in Section 9.6.3 Adding to the Natural Hierarchy you will capture the following hierarchy tree.



To get a more compressed tree (vertically) you may follow the `'h'` with a `'-'`. This results in a tree where every other line is removed. For example, if you had used the option `+vh-m` the same tree would appear as:



The characters used to draw the hierarchy may be the PC graphics characters (as shown) or may be regular ASCII characters. On the PC the former is the default. To obtain ASCII, turn off the Hierarchy Graphics Flag with the `-fhg` option.

9.7 Hints on Strong Typing

1. Beware of excessive casting. If, in order to pull off a system of strong typing you need to cast just about every access, you are missing the point. The casts will inhibit even ordinary checking, which has considerable value. Remember, strong type-checking is gold, normal type-checking is silver, and casting is brass.

2. Rather than cast, use type hierarchies. For example:

```
/*lint -strong(AXJ,Tight) -strong(,Loose) */
typedef int Tight;
typedef Tight Loose;
```

   `Tight` has a maximal amount of Strong Type checking; `Loose` has none. Since `Loose` is defined in terms of `Tight` the two types are interchangeable from the standpoint of Strong Type checking. Presumably you work with `Tight int`'s most of the time. When absolutely necessary to achieve some effect, `Loose` is used.

3. A time when it is really good to cast is to endow some otherwise neutral constant with a special type. `FZERO` in **Section 9.6.2 "The Natural Type Hierarchy"**.

4. For large, mature projects enter strong typing slowly working on one family of strong types at a time. A family of strong types is one hierarchy structure.

5. Don't bother with making pointers to functions strong types. For example:

```
typedef  int (*Func_Ptr)(void);
```

   If you make `Func_Ptr` strong, you're not likely to get much more checking that if you didn't make it strong. The problem is that you would then have to cast any existing

function name when assigning to such a pointer. This represents a net loss of type-checking (remember: gold, silver, brass).

6. Rather than making a pointer a strong type, make the base type a strong type. For example:

```
typedef char TEXT;
typedef TEXT *STRING;

TEXT buffer[100];
STRING s;
```

It may seem wise to strong type both **STRING** and **TEXT**. This would be a mistake since whenever you assign **buffer** to **s**, for example, you would have to cast. But note that **-strong(Ac,STRING)** would allow the assignment. It is usually better to strong type just **TEXT**. Then when **buffer** is assigned to **s** the indirect object **TEXT** is strongly checked and no cast is needed.

This holds for structures as well as for scalars. For example, in MS Windows programming there are a number of **typedef**'ed types that are pointers. Examples include: **LPRECT**, **LPLOGFONT**, **LPMSG**, **LPSTR**, **LPWNDCLASS**, etc. If you make these **-strong(A)** you will have problems passing to Windows functions, the addresses of a Window's **struct**. At most make them **-strong(AcX)**.

7. Care is needed in declaring strong self-referential **struct**'s. The usual method, i.e.,

```
typedef struct list { struct list * next ; ... }
LIST;
```

is incompatible with making **LIST** a strong type because its member **next** will not be a pointer to a strong type. It is better to use:

```
typedef struct list LIST;
struct list { LIST * next; ...};
```

This is explicitly sanctioned in ANSI/ISO C (see 3.5.2.3 of **[2]**) and will make **next** compatible with other pointers to **LIST**.

8. Once a type is made strong it should not then be made unstrong or weak. For example

```
//lint  -strong(AJX)
typedef  int INT;

//lint  -strong(,INT)

// INT is still strong
```

9.8 Reference Information

Strong Expressions

An expression is strongly typed if

1) it is a strongly typed variable, function, array, or member of **union** or **struct** or a dereferenced pointer to a strong type.

2) it is a cast to some strong type.

3) it is one of the ***type-propagating*** unary operators, (viz. **+ - ++ -- ~**), applied to a strongly typed expression.

4) it is formed by one of the ***balance and propagate*** binary operators applied to two strongly typed expressions (having the same strong type). The balance and propagate operators consist of the five binary arithmetics (**+ - * / %**), the three bit-wise operators (**& | ^**), and the conditional operator (**? :**).

5) it is formed by a multiplication operator where at least one of the arguments is a dimension.

6) it is a shift operator whose left side is a strong type.

7) it is a comma operator whose right side is a strong type.

8) it is an assignment operator whose left side is a strong type.

9) it is a ***Boolean operator*** and some type has been designated as Boolean (with a **b** or **B** flag in the **-strong** option). The Boolean operators consist of the four relationals (**> >= < <=**), the two equality operators (**== !=**), the two logical operators (**|| &&**), and unary **!**

General Information

When the option

-strong(***flags*** **[,** ***name*** **]** ... **)**

is processed, ***name*** and ***flags*** are entered into a so-called Strong Table created for this purpose. If there is no ***name***, then a variable, Default Flags, is set to the ***flags*** provided. When a subsequent **typedef** is encountered within the code, the Strong Table is consulted first. If the **typedef** name is not found, the Default Flags are used. These flags become the identifying flags for strong typing purposes for the type.

The option

```
-index( flags, ixtype, sitype[, ...] )
```

is treated similarly. Each ***sitype*** is entered into the Strong Table (if not already there) and its index flags OR'd with other strong flags in the table. A pointer is established from ***sitype*** to ***ixtype***, which is another entry in the Strong Table.

For these reasons it does not, in general, matter in what order the **-strong** options are placed other than that they be placed before the associated **typedef**. There should be at most one option that specifies Default Flags.

9.9 Strong Types and Prototypes

If you are producing prototypes with some variation of the **-od** option (Output Declarations), and if you want to see the **typedef** types rather than the raw types, just make sure that the relevant **typedef** types are strong. You can make them all strong with a single option: **-strong()**. Since you have not specified '**A**', '**J**' or '**X**' you will not receive messages owing to strong type mismatches for Assigning, Joining or eXtraction. However, you may get them for declarations. You can set

```
-etd(strong)
```

to inhibit any such messages or better, correct the inconsistencies.

10. VALUE TRACKING

10.1 Initialization Tracking

An historic forerunner of Value Tracking is tracking the initialization (or lack thereof) of auto variables.

This section has to do with messages **644**, **645** ("may not have been initialized"), **771**, **772** ("conceivably not initialized"), **530** ("not initialized"), and **1401** - **1403** ("member ... not initialized"). These messages refer to **auto** variables and data members while in constructors.

For example, given the code

```
if( a ) b = 6;
else c = b;
a = c;
```

assume that neither **b** nor **c** were previously initialized. PC-lint/FlexeLint reports that **b** is not initialized (when its value is assigned to **c**) and that **c** may not have been initialized (when its value is assigned to **a**).

In earlier versions (and in conventional lint's) a single unintelligent sweep is taken, which would regard **b** and **c** as having been initialized before use.

while loops and **for** loops are not quite the same as **if** statements. Consider, for example, the following code:

```
while ( n-- )
    {
    b = 6;
    ...
    }
c = b;
```

assuming that **b** had not been initialized earlier, we report that **b** is "conceivably not initialized" when assigned to **c** and give a lighter Informational message. The reason for distinguishing this case from the earlier one is that it could be that the programmer knows the body of the loop is always taken at least once. By contrast, in the earlier case involving **if** statements, the programmer would be hard-pressed to say that the **if** condition is always taken, for that would imply, at the least, some redundant code, which could be eliminated.

The **switch** is more like an **if** than a **while**. For example,

```
switch ( k )
```

```
    {
case 1:  b = 2; break;
case 2:  b = 3;
          /* Fall Through */
case 3:  a = 4; break;
default:  error();
    }
c = b;
```

Although **b** has been assigned a value in two different places, there are paths that might result in **b** not being initialized. Thus, when **b** is assigned to **c** a possibly-uninitialized message is obtained. To fix things up you could assign a default value to **b** before the **switch**. This quiets lint but then you lose the initialization detection in the event of subsequent modifications. A better approach may be to fix up the **case**'s for which **b** has not been assigned a value. We will show this below.

If the invocation of **error()** is one of those instances which "can't occur but I'll report it anyway," then you should let PC-lint/FlexeLint know that this section of code is not reachable. If **error()** does not return, it should be marked as not returning by using the option

```
-function(exit,error)
```

This transfers the special property of the **exit** function to **error**. Alternatively, you may mark the return point as unreachable as shown in the following fixed up example:

```
switch ( k )
    {
case 1:  b = 2; break;
case 2:
case 3:  b = 3; a = 4; break;
default:  error();
           /*lint -unreachable */
    }
c = b;
```

Don't make the mistake of placing the **-unreachable** directive before the call to **error()** as this property is not transmitted across the call. If there is a **break** after the call, make sure the directive is placed ***before*** the **break**. Code after a **break**, is never considered reachable, so the directive placed after the **break** would have no effect.

Another way to get the "not initialized" message is to pass a pointer variable to **free** (or to some function like **free** -- See Section 11.1 Function Mimicry (-function)). For example:

```
if( n ) free( p );
...
p->value = 3;
```

will result in **p** being considered as possibly not initialized at the point of access.

Forward **goto**'s are supported in the sense that the initialization state of the **goto** is merged with that of the label. Thus, if **b** is not yet initialized, the code:

```
if ( a ) goto label;
b = 0;
label: c = b;
```

will receive a possibly-uninitialized message when **b** is assigned to **c**. However, backward **goto**'s, since they do not reduce the initialization state, are ignored.

When the checking for possibly uninitialized variables is first applied to a large mature project, there will be a small number of false hits. Experience indicates that they usually involve code that is not especially well structured or may involve some variation of the following construct:

```
if( x ) initialize y
...
if( x ) use y
```

For these cases simply add an initializer to the declaration for **y** or use the option **-esym(644,y)**.

Data members are considered to be initially uninitialized in constructors. For example:

```
class X
     {
     // Warnings 1401 and 1402
     X() { if(n) n = 3; }
     int  n;
     int m;
     };
```

results in the message **1402** that **n** is not initialized when it is accessed and message **1401** that **m** has not been initialized by the constructor. Calling a member function (that is not **const**) changes things:

```
class X
    {
    // the call to f() inhibits initialization warnings
    X() { f(); if( n )  n = 3; }
    void f();
    int n;
    int m;
```

```
    };
```

If `X::f()` was `const` or if some non member function were called instead, no assumptions about initialization would be made:

```
class X
    {
    // Warnings 1401 and 1402 are issued
    X() { g(); if( n ) n = 3; }
    void g() const;
    int n;
    int m;
    };
```

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.

10.2.1 The assert remedy

It is possible to obtain spurious messages (false hits) that can be remedied by the judicious use of **assert**. For example

```
char buf[4];
char *p;

strcpy( buf, "a" );
p = buf + strlen( buf );  // p is 'possibly' (buf+3)
p++;                      // p is 'possibly' (buf+4)
*p = 'a';      // Warning 661 - possible out-of-bounds reference
```

PC-lint/FlexeLint is not aware in all cases what the true values of variables are. In the above case, a warning is issued where there is no real danger. You can inhibit this message directly by using

```
      *p = 'a';     //lint !e661
```

Alternatively, you can use your compiler's **assert** facility as in the following:

```
#include <assert.h>
      ...
char buf[4];
char *p;

strcpy( buf, "a" );
p = buf + strlen( buf );
assert( p < buf + 3 );   // p is 'possibly' (buf+2)
p++;                     // p is 'possibly' (buf+3)
```

```
*p = 'a';                // no problem
```

Please note that **assert** will become a no-op if **NDEBUG** is defined so make sure this is not defined for linting purposes.

In order for the **assert()** facility to have this effect with your compiler's **assert.h** it may be necessary to include an option within your compiler options file. For example, suppose that the **assert** facility is implemented via a macro defined by your compiler as

```
#define assert(p) ((p) ? (void)0 : __A(…))
```

Presumably **__A()** issues a message and doesn't return. It would then be necessary to give the option

```
-function( exit, __A )
```

which transfers the no-return property of **exit** to the **__A** function (See Section 11.1 Function Mimicry (-function)).

Alternatively, your compiler may implement **assert** as a function. For example:

```
#define assert(k) _Assert(k,…)
```

To let PC-lint/FlexeLint know that **_Assert** is the **assert** function you may copy the semantics of the **__assert()** function (defined in Section 11.1 Function Mimicry (-function)) using the option

```
-function( __assert, _Assert )
```

or

```
-function( __assert(1), _Assert(1) )
```

The latter form can be employed for arguments other than the first.

For many compilers, the necessary options have already been placed in the appropriate compiler options file.

In the event that there is no obvious solution that you can employ with your compiler's **assert.h**, you can always place an **assert.h** in your PC-lint/FlexeLint directory (where presumably file searching is directed with a **-i** option and which takes precedence over your compiler's header files).

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.

10.2.2.1 Specific Calls - Specific Walks

When a function call is encountered, information about the arguments in the call is determined. The information includes the size of arrays, the value of integers, the nullness of pointers, etc. This is referred to as a ***Specific Function Call*** or ***Specific Call*** for short. Each unique such call is saved in a list associated with the called function. You can monitor the accumulation of such calls using the verbosity option **-vc**. (You would normally combine this with at least module verbosity so you might actually write **-vcm**.)

When a definition for a function is encountered, it is first processed in what is called ***General Processing***. General Processing (sometimes referred to as the ***General Walk***) will do syntax checking and will do a limited amount of value tracking. It will not assume the arguments to have any specific values owing to any prior function calls. General Processing will produce an internal tree.

After General Processing, there will be a series of ***Specific Function Call Walks*** (***Specific Walk*** for short) one for each Specific Call that is currently in the list for that function.

At the beginning of each Specific Walk, the function's parameters are initialized with the values implied by the Specific Call. You can get a verbosity message at each such Specific Walk using **-vw**. The walk includes a similar kind of value tracking that takes place in General Processing. But since the tree that is being walked has already been produced, there is no macro processing, no template expansion and no syntax analysis during a Specific Walk. There is also an automatic inhibition of certain messages over the course of the walk. This is because some messages are inappropriate at this time. For example, consider the test:

```
if( a == 0 ) ...
```

If **a** were a parameter, then during a Specific Walk we may be given that its value was 0 but it would be inappropriate to comment (Message **774**) that the condition would *always* be true since it's only true of this specific call.

At each return statement within the Specific Walk, information is tucked away in a return value associated with this Specific Call. When this Specific Call is again seen during General Processing or a Specific Walk of this or another function, the return value can be used as the resulting value.

10.2.2.2 Error inhibition

To suppress messages on a per-statement basis in specific walks be sure to use the curly brace form. For example:

```
int f( int k )
    {
    /*lint -e{414} inhibit divide by 0 message */
    return 100/k;
    }
```

will cause Message **414** to be inhibited when processing the return statement not only during the General Walk but in all Specific Walks as well. See also Section 5.2. Error Inhibition Options. Note that the one line error suppression (**!e**...) is not effective in Specific Walks.

10.2.2.3 Specific Call Portrayal

A Specific Call is portrayed during verbosity messages and as a prologue to messages given during a Specific Walk. For example, in the output of processing **t4.cpp** we see:

```
During Specific Walk:
    File t4.cpp line 3:  f(2)
        ...
```

The **f(2)** is a Specific Call portrayal.

The portrayal consists of a function name followed by a parenthesized list of arguments. The arguments are portrayed as follows:

First, any argument portrayed must be associated with an explicit parameter. Arguments associated with ellipsis (...) are ignored. For example, given:

```
void f( int, … );
 ...
    f(10, 10);
```

then the call to **f()** is portrayed as **f(10)**.

In the case (only legal in C) where a prototype is not explicitly given, an effective prototype for this purpose will be created.

Integer Portrayal

Integer parameters whose values are known are represented in their conventional decimal representation, including a sign. If the value is not known but there is a possibility that it might have a particular value, then the value is followed by a '**?**'. For example:

```
void g(int n)
    {
    if ( n == 4 ) h();
    f(n);
    ...
```

then the call to **f()** is portrayed as **f(4?)**.

There are times when the only thing known about an argument is that it does not equal 0. For example:

```
void g( int n)
    {
    if(n) f(n);
    ...
```

then the call to **f()** is portrayed as **f(!=0)**.

Alternative values can also be portrayed. For example:

```
void g( int  b)
    {
    int n = 0;
    if ( b ) n = 4;
    f( n );
    ...
```

then the call to **f()** will be portrayed as

```
f( 4? | 0? )
```

Pointer and Array Portrayal

An array is portrayed as

[*n*]

where ***n*** is its dimension. The ***n*** is in terms of elements not bytes. When arrays are passed as arguments, they are, of course, converted to pointers and so it follows that pointers are represented as arrays of length equal to the number of elements being pointed to. Thus:

```
void g()
    {
    int a[10];
    int p = a+1;
    f( p );
    ...
```

then the call to **f()** is portrayed as **f([9])**. Considerations regarding doubtful values (the '**?**' suffix), non-zero values (**!=0**) and ambiguous values (using **'|'** notation) apply to pointers and arrays as well as to integers.

Finally, when nothing at all is known about an argument the notation **?** is used.

10.2.2.4 Specific Walk Options Summary

The following set of options relate to the control of interfunction value tracking.

-fsp is a flag that can turn off both the accumulation of Specific Calls and the taking of Specific Walks.

-format_specific The prologue to a Specific Walk error message is controlled by this option. Its default value is:

```
-"format_specific=\nDuring Specific Walk:\n%{   File %f line %l: %i\n%}"
```

There are two new escapes:

%{...%} = designates a repetition, one for each invocation in the chain of calls leading to the Specific Walk.
%i = designates a call Invocation. This differs from the **%i** used in other format strings in that an argument list is appended.

-passes(*k* [*,Options1* [*,Options2*]])This option allows multiple passes over the set of modules provided as input. For example:

```
-passes(2)
```

requests two passes (the default is 1). The second and third parameters (***Options1*** and ***Options2***) can be used to provide options that are used only for the 2nd and subsequent passes. These are described more fully below.

The primary purpose of the multiple pass option is to support interfunction value tracking to a depth greater than what can be supported in a single pass. Other uses of the multi-pass option may present themselves in the future.

At the start of each pass there is an attempt to reset, insofar as is reasonably practical, all the options to their original setting. Then all options are serially reprocessed for each pass in the same order as in the first pass. This includes reexamining the environment variables LINT and INCLUDE for option implications. It also includes fully processing any indirection file (**`*.lnt`** file) since these may contain options or file names.

Most options internally represent parameters which are simply reset at the beginning of each pass. For example, error suppression and flags are reset to their original values and the options are simply reprocessed in every particular.

For some options it seemed inappropriate to reset to a ground condition and so these options are ignored in 2nd and subsequent passes, while the information they represent is left intact. These include those involved with strong typing: **`-father`**, **`-index`**, **`-parent`**, and **`-strong`**; those involved with function semantics: **`-function`**, **`-printf`**, **`-scanf`**, **`-sem`**, **`-wprintf`**, and **`-wscanf`**; and a few others which are inappropriate to a multi-pass setting and include: **`-lobbase`**, **`+macros`**, and the size options (**`-s`**...). However, any option explicitly placed in the 2nd or 3rd arguments to the **`-passes`** option (or embedded in a file placed there) will not be ignored.

Another issue in designing a multi-pass operation is that of error inhibition. Presumably, most warnings seen on the first pass should not be displayed on the second (and subsequent) passes because it just adds to clutter. For this reason, during pass 2 (and subsequent passes), just before the first module is processed, that is, after the user has established his (or her) error suppression strategy, PC-lint/FlexeLint inhibits most messages. It inhibits all but a select group as described below in **`-specific`**.

If this is not the desired behavior, the user can alter, replace or cancel this error suppression to reflect his own intentions. He does this through ***`Options1`*** and ***`Options2`***, the second and third options to the **`-passes`** option. ***`Options1`*** is processed just before the error suppression described above (only during pass 2 and subsequent passes) and ***`Options2`*** is processed just after. ***`Options1`*** is normally null. For example:

```
-passes(4,,+e551)
```

requests 4 passes and will accept the inhibition but will enable message **`551`**, which would normally be disabled in the 2nd and subsequent passes.

The whole purpose of ***`Options1`*** is to allow the user to issue a **`-save`** as in:

```
-passes( 3, -save, -restore -dPASS2 )
```

requests 3 passes and will issue a **-save** option just before the error suppression and a **-restore** just after. The effect is to cancel out our error suppression. It also defines a symbol that presumably will be defined only in passes 2 and later.

You may also specify a file as an argument.

```
-passes(2,,pass2.lnt)
```

causes the indirection file **pass2.lnt** to be processed only for pass 2 and beyond.

Even though you may have a complex multi-pass strategy established in some indirection file you may still change just the number of passes on the command line. For example, suppose you place in **std.lnt** the option:

```
-passes(1,-save,-restore pass2.lnt)
```

If you then place the option **-passes(3)** (or equivalently **-passes=3**) on the command line after **std.lnt**, you will change the number of passes to 3 but you will not change ***Options1*** or ***Options2*** so your multi-pass strategy will stay intact. This will permit you to experiment with more or fewer passes.

-specific(*Options1* [*, Options2*])

allows the user to specify options just before the start of each Specific Walk (***Options1***) and just after finishing each walk (***Options2***). For example:

```
-specific( -e413 )
```

will inhibit Warning **413** during each Specific Walk. The warning will automatically be restored after each walk is over. The sequence of events is as follows.

1) The error suppression state is saved.
2) All but a select group of messages are suppressed.
3) ***Options1*** is processed.
4) The Specific Walk is made.
5) ***Options2*** is processed.
6) The error suppression state is restored to its original.

Therefore ***Options2*** is normally null. It could be used to restore a flag as in:

```
-specific( -e413 ++flb, --flb)
```

Note that multiple options can be given as a single argument. Also a **.lnt** file could be provided as argument.

At this writing the select group of messages **not** suppressed in step 2, consists of messages **413**, **414**, **415**, **416**, **418**, **419**, **420**, **422**, **426**, **428**, **429**, **432**, **433**, **454**, **455**,

520, **521**, **522**, **591**, **613**, **661**, **662**, **668**, **669**, **670**, **671**, **676**, , **794**, **795**, **796**, **797**, **802**, **803**, **804**, **807**, **817**, **826**, **830**, **831**, and **948** and all Internal and Fatal errors. Effectively, for each number not in the select group an option of **-e#**, where # is the message number, is issued.

-specific_climit(*n*) specifies a Call limit. The total number of Specific Calls recorded for any one function is limited to ***n***. Because of recursion, the total number of Specific Calls made on any one function can be huge. This option prevents any one function from hogging resources. By default, the value is 0, implying no limit.

-specific_retry(*n*) indicates whether Specific Calls walked in one pass, are rewalked on subsequent passes. The parameter ***n*** can be either 0, meaning no, or 1 meaning yes. The default is 1.

You might want to set this value to 0 if you feel that rewalking a Specific Call will not yield much additional information.

It may seem wasteful to rewalk the same Specific Call on each pass but each call can turn up new information and can alter the return value associated with that call.

For example, consider the following two-function module:

```
int fact(int n)
    {
    if( n == 0 ) return 1;
    else return n * fact(n-1);
    }

int a()
    {
    return 100 / (6 - fact(3));
    }
```

The first function, **fact(n)**, computes **n** factorial (**n!**) recursively. Since **fact(3)** is indeed equal to 6, function **a()** has a divide-by-zero error. To find that error, a Specific Call to **fact(3)** results in a Specific Walk in pass 2 that turns up a Specific Call to **fact(2)**. The Specific Walk of **fact(2)** turns up a Specific Call to **fact(1)**. But no return values for **fact(3)**, **fact(2)**, or **fact(1)** are yet available. It is not until several passes later that a Specific Walk of **fact(3)** learns that the return value of **fact(2)** is 2 at which time it can compute the return value of **fact(3)**. On that pass the divide-by-zero error is detected. If functions were not rewalked with the same specific arguments, this condition could not have been detected.

It is useful to track the accumulation of Specific Calls and the taking of Specific Walks using the verbosity options **-vcw**... (See **-vc** and **-vw** below.)

-specific_wlimit(*n*) specifies a Walk limit. At the end of the General Walk any Specific Call that had not earlier been walked (in some prior pass) will result in a Specific Walk. These Specific Walks can themselves generate additional Specific Calls. If the calls are recursive calls and if all these calls are walked, the situation can result in a combinational explosion. For this reason the total number of Specific Walks is limited to the number of Specific Calls on hand at the completion of the General Walk plus this value ***n***. By default ***n*** is 1. This allows for the walking of one additional recursive call. For example, consider:

```
void f( int n ) { f( n+1 ); }
```

If a Specific Call of **f(1)** had been earlier detected then this will result in a Specific Walk of **f(1)**, which will generate the Specific Call **f(2)**, which will result in a Specific walk of **f(2)**, which will generate the Specific Call of **f(3)**, which won't be walked, this pass, because the limit is 1. It will be walked in the next pass.

If you make the limit negative, this will be taken as no limit and you will get the maximum recursive impact. ***Caution***: this could cause a nonterminating loop.

-vc a Verbosity flag to issue a report whenever a new Specific Call is encountered.

-vw a Verbosity flag to issue a report whenever a Specific Walk is initiated.

10.2.3 Tracking Static Variables

Version 9.0 of PC-lint/FlexeLint is introducing static variable tracking. By static variables we mean variables having static storage duration. This includes not only nominally static variables such as static variables local to a module or local to a function but also global variables and, in C++, variables at namespace scope and variables that are static members of a class.

Consider the following simple program. It clearly has a divide by 0 problem.

```
sl.cpp:
     // Example 1
     //lint -passes(3) -vc -e831 -e714
     int n;
     int f() { return 10 / (n-1); }
     int main() { n = 1; return f(); }
```

When run through PC-lint/FlexeLint the output is:

```
--- Module:  s1.cpp (C++)
    --- Logging Specific Call:  f() #1

/// Start of Pass 2 ///
```

```
--- Module:  s1.cpp (C++)
    --- Logging Specific Call: f() n=1 #2

/// Start of Pass 3 ///

--- Module:  s1.cpp (C++)

During Specific Walk:
   File s1.cpp line 5:  f() n=1 #2
s1.cpp(4) : Warning 414:  Possible division by 0 [Reference: ...
```

Sure enough, the divide by 0 is detected; but note that the message does not appear until Pass 3. The reason for this can be gleaned from some of the verbosity information. Recall that **`-vc`** causes a report to be issued whenever a Specific Call is logged. In that report, argument values appear, and, for static variable tracking, each variable together with its current value is written. During Pass 1 there is no tracking of static variables since we do not yet know which function calls can modify which variables. This information is not complete until all functions have been seen.

So in this example the first time a Specific Call is recorded that contains any static variable information is in Pass 2. Here we note the call from **`main()`** to **`f()`** at which time **`n`** is **`1`**. But since the definition of **`f()`** has already passed we do not issue the diagnostic until we encounter that definition in Pass 3.

Consider a second example where **`main()`** does not call **`f()`** directly but, rather, calls an intermediate function **`g()`** which in turn calls **`f()`**.

```
s2.cpp:
    // Example 2
    //lint -passes(4) -static_depth(2) -vc -e831 -e714
    int n;
    int f() { return 10 / (n-1); }
    int g() { return f(); }
    int main() { n = 1; return g(); }
```

Here we have a new problem. By default **`g()`** knows nothing of the variable **`n`** because the variable is not used by **`g()`**. Variable **`n`** is said not to be in the zone of **`g()`**. By default, the static depth is 1. But by increasing the static depth to 2 via the option **`-static_depth(2)`** (shown in the example) the zone of each function is extended to include all the variables directly used by all the functions directly called by that function. In particular, the zone of **`g()`** now includes variable **`n`**. There is no intrinsic limit to the value that can be used to indicate the static depth. See **`-static_depth(n)`** in **Section 5.7 Other Options**.

The output of PC-lint/FlexeLint is shown below and again it is instructive.

```
--- Module:  s2.cpp (C+++
    --- Logging Specific Call:  f() #1
```

```
    --- Logging Specific Call:  g() #1

/// Start of Pass 2 ///
--- Module:  s2.cpp (C+++
    --- Logging Specific Call:  f() #2
    --- Logging Specific Call:  g() n=1 #2

/// Start of Pass 3 ///
--- Module:  s2.cpp (C+++
    --- Logging Specific Call:  f() n=1 #3

/// Start Pass 4 ///

--- Module:  s2.cpp (C++)

During Specific Walk:
    File s2.cpp line 6:  g() n=1 #2
    File s2.cpp line 5:  f() n=1 #3
s2.cpp(4) : Warning 414:  Possible division by 0 [Reference: ...
```

Again, Warning **414** indicates a division by 0 but this time it waits until Pass 4. As we can see from the output, Specific Calls are logged in Pass 1 but by construction no static variables are part of the call information. In Pass 2 a call to **f()** is recorded from **g()** but at this time **g()** had no reason to believe that **n** had any particular value. But **main()** creates a Specific Call to **g()** (labeled #2) that indicates that **n** has the value of 1. In Pass 3, Specific Call #2 to **g()** is walked resulting in a new Specific Call to **f()** (labeled #3) which has **n** set to 1. Finally, by Pass 4, a Specific Walk to **f()** is triggered which notes the misfortunate value of 1 for variable **n**.

There is clearly a tradeoff between performance and information obtained. By increasing static depth and/or by increasing the number of passes your analysis is more penetrating but the time spent and the storage consumed will be greater. By increasing static depth you can greatly increase storage requirements because a new Specific Walk will be recorded for each call that differs from some other call by only one value of some obscure global variable.

You can monitor the storage consumed by using the '**s**' verbosity flag (as in **-vs**)

The reason that tracking static variables begins in pass 2 is that we need to know which function calls can potentially modify which static variables, and this analysis cannot be made until the function definitions have been seen.

If a function call can result, to whatever depth, in a modification of a variable, our certainty of the value of that variable is diminished. This potential for modification depth is unlimited and has nothing to do with the static depth.

11. SEMANTICS

11.1 Function Mimicry (`-function`)

This section describes how some properties of built-in functions can be transferred to user-defined functions by means of the option **`-function`**. See also **`-printf`** and **`-scanf`** in Section 5.7 Other Options. See also Section 11.2 Semantic Specifications to see how to create custom function semantics.

11.1.1 Special Functions

PC-lint/FlexeLint is aware of the properties (which we will call semantics) of some standard functions, which we refer to as special functions. A complete list of such functions is shown in Section 11.1.2 Function listing.

For example, function **`fopen()`** is recognized as a special function. Its two arguments are checked for the possibility of being the NULL pointer and its return value is considered possibly NULL. Similarly, **`fclose`** is regarded as a special function whose one argument is also checked for NULL. Thus, the code:

```
if( name ) printf ( "ok\n" );
f = fopen( name, "r" );         // Warning! name may be NULL
fclose ( f );                   // Warning! f may be NULL
```

will be greeted with the diagnostics indicated. You may transfer all three semantics of **`fopen`** to a function of your own, say **`myopen`**, by using the option

```
-function( fopen, myopen )
```

Then, PC-lint/FlexeLint would also check the 1st and 2nd arguments of **`myopen`** for NULL and assume that the returned pointer could possibly be NULL. In general, the syntax of **`-function`** is described as follows:

`-function(` ***`Function0`*****`,`** ***`Function1`*** [***`, Function2`***] … **`)`**

specifies that ***`Function1`***, ***`Function2`***, etc. are like ***`Function0`*** in that they exhibit special properties normally associated with ***`Function0`***.

The arguments to **`-function`** may be subscripted. For example, if **`myopen`** were to check its 2nd and 3rd arguments for NULL rather than its 1st and 2nd we could do the following:

```
-function( fopen(1), myopen(2) )
```

```
-function( fopen(2), myopen(3) )
```

This would transfer the semantics of NULL checking to the 2nd and 3rd arguments of **myopen**. This could be simplified to

```
-function( fopen(1), myopen(2), myopen(3) )
```

since the property of **fopen(1)** is identical to that of **fopen(2)**. Any previous semantics associated with the 2nd and 3rd arguments to **myopen** would be lost.

To transfer the return semantics you may use the option

```
-function( fopen(r), myopen(r) )
```

Some functions have a semantic that is not decomposable to a single argument or return value but is rather a combined property of the entire function. For example

```
char * fread( char *, size_t, size_t, FILE * );
```

has, in addition to the check-for-NULL semantics on its 1st and 4th arguments, and the check-for-negative semantics on the 2nd and 3rd arguments, an additional check to see if the size of argument 2 multiplied by argument 3 exceeds the buffer size given as the 1st argument. This condition is identified as semantic **S6** in Section 11.1.2 Function listing. Thus

```
char buf[100];
fread( buf, 100, 2, f );        // Warning
```

To transfer this function-wide property to some other function we need to use the 0 (zero) index. Thus

```
-function( fread(0), myread(0) )
```

will transfer just the overflow checking (**S6** as described above) and not the argument checking. That is, of the semantics appearing in Section 11.1.2 Function listing for row labeled **fread**, the semantics transferred are only those associated with column **f(0)** and not with other columns.

As a convenience, the subscript need not be repeated if it is the same as the 1st argument. Thus

```
-function( fread(0), myread )
```

is equivalent to the earlier option.

Just as in the case of **fopen** you may transfer all the properties of **fread** to your own function by not using a subscript as in:

```
-function( fread, myread )
```

You may remove any or all of these semantics from a special function by not using a 2nd argument to the **-function** option. Thus

```
-function( fread )
```

will remove all of the semantics of the **fread** function and

```
-function( fread(0) )
```

removes only the special semantics described above.

In summary, the semantics of functions is conceptually retained in a 2 dimensional (sparse) array as depicted in Section 11.1.2 Function listing. An option of the form

```
-function( function(index), ... )
```

copies a single semantic into a destination or destinations. An option of the form

```
-function( function, ... )
```

copies a whole row of semantics.

You may transfer semantics to member functions as well as non-member functions. Thus

```
-function( exit, X::quit )
```

transfers the properties of **exit()** to **X::quit()**. The semantics in this case is simply that the function is not expected to return. Please note that this option must be given before the definition of class **X**. (Actually, it need only be given before the closing brace that defines class **X**.) This is usually not a problem as such options are normally provided in a separate **.lnt** file preceding all modules; but such options may also be given in a 'lint' comment at the class definition.

As another example involving member functions consider the following:

```
//lint -function( strlen(1), X::X(1), X::g )
//  both X::X() and X::g() should have their 1st
//  argument checked for NULL.
//lint +fpn pointer parameters may be NULL

class X
    {
    public:
    char *buf;
    X( char * );
```

```
    void g( char * );
    };

void f( char *p )   // p may be NULL because of +fpn
    {
    X x(p);         // Warning 668
    x.g( p );
    }
```

In this example, the semantics associated with the 1st argument of **strlen** are transferred to the 1st argument of **X::X** and to the 1st argument of **X::g**. As the example illustrates, when we speak of the *n*th argument passed to a member function we are ignoring in our count the implicit argument that is a pointer to the class (this is always checked for **NULL**).

No distinction is made among overloaded functions. Thus, if **X::X(int *)** is checked for **NULL** then so is **X::X(char *)**. If there is an **X::X(int)** then its argument is not checked because its argument is not a pointer. If there is an **X::X(int *, char *)** then the 1st argument is checked, but not the 2nd.

We support two forms of the **assert** function. **__assert** (double underscore **assert**) and **___assert** (triple underscore **assert**) The latter differs from the former in that it always returns.

For example, suppose you have:

```
...
if( some_condition )
      {
      ASSERT( false );
      // do some cleanup
      }
...
```

Assume further that you are using **lib-mfc.lnt** and so the **ASSERT** is defined to be **__assert**, the two underscore version. Then, given that we can deduce that the argument is always false, we must then deduce that it does not return. But then, we report that the cleanup code is unreachable. This could readily be handled by using the **-unreachable** option but if the practice is widely used, it can be a bit of a chore to insert this option in many different locations.

If you were to change all instances of **__assert** within **lib-mfc.lnt** to **___assert** the problem would be solved at least until the next time you upgraded **lib-mfc.lnt**. A perhaps better approach is to transfer the return semantics of the triple underscore variety to the double underscore version as follows:

```
-function( ___assert(r), __assert(r) )
```

11.1.2 Function listing

Function	f(0)	f(r)	f(1)	f(2)	f(3)	f(4)	f(5)
___assert	-	-	S20	-	-	-	-
__assert		-	S20	-	-	-	-
abort	S3	-	-	-	-	-	-
asctime	-	-	S1	-	-	-	-
atof	-	-	S27	-	-	-	-
atoi	-	-	S27	-	-	-	-
atol	-	-	S27	-	-	-	-
bsearch	-	S2	S1	S1	S21	S21	S1
calloc	S4	S2	S21	S21	-	-	-
clearerr	-	-	S1	-	-	-	-
ctime	-	-	S1	-	-	-	-
exit	S3	-	-	-	-	-	-
fclose	-	-	S19	-	-	-	-
feof	-	-	S1	-	-	-	-
ferror	-	-	S1	-	-	-	-
fgetc	-	-	S1	-	-	-	-
fgetpos	-	-	S1	S1	-	-	-
fgets	S5	S2	S27	S21	S1	-	-
fopen	-	S2	S27	S27	-	-	-
fprintf	-	-	S1	S27	-	-	-
fputc	-	-	-	S1	-	-	-
fputs	-	-	S27	S1	-	-	-
fread	S6	-	S1	S21	S21	S1	-
free	-	-	S7	-	-	-	-
freopen	-	S2	S1	S1	S1	-	-
frexp	-	-	-	S1	-	-	-
fscanf	-	-	S1	S1	-	-	-

Function	f(0)	f(r)	f(1)	f(2)	f(3)	f(4)	f(5)
fseek	-	-	S1	S21	-	-	-
fsetpos	-	-	S1	S1	-	-	-
ftell	-	-	S1	-	-	-	-
fwrite	S8	-	S1	S21	S21	S1	-
getc	-	-	S1	-	-	-	-
getenv	-	-	S1	-	-	-	-
gets	S18	S2	S27	-	-	-	-
gmtime	-	S2	S1	-	-	-	-
localtime	-	-	S1	-	-	-	-
longjmp	S3	-	S1	-	-	-	-
malloc	S9	S2	S21	-	-	-	-
mbstowcs	-	-	S27	S27	-	-	-
mbtowc	-	-	S1	-	-	-	-
memchr	S10	S2	S1	-	S21	-	-
memcmp	S11	-	S1	S1	S21	-	-
memcpy	S12	-	S1	S1	S21	-	-
memmove	S12	-	S1	S1	S21	-	-
memset	S13	-	S1	-	S21	-	-
mktime	-	-	S1	-	-	-	-
modf	-	-	-	S1	-	-	-
operator new	S9	-	S21	-	-	-	-
operator new[]	S9	-	S21	-	-	-	-
operator delete	-	-	S29	-	-	-	-
operator delete[]	-	-	S30	-	-	-	-
perror	-	-	S1	-	-	-	-
printf	-	-	S27	-	-	-	-
putc	-	-	-	S1	-	-	-

Function	f(0)	f(r)	f(1)	f(2)	f(3)	f(4)	f(5)
puts	-	-	S27	-	-	-	-
qsort	-	-	S1	S21	S21	S1	-
realloc	S23	S2	S24	S21	-	-	-
remove	-	-	S27	-	-	-	-
rename	-	-	S27	S27	-	-	-
rewind	-	-	S1	-	-	-	-
scanf	-	-	S27	-	-	-	-
setbuf	-	-	S1	-	-	-	-
setjmp	S14	-	-	-	-	-	-
setvbuf	-	-	S1	-	-	-	-
sprintf	-	-	S27	S27	-	-	-
sscanf	-	-	S27	S27	-	-	-
strcat	S15	-	S27	S27	-	-	-
strchr	-	S2	S28	-	-	-	-
strcmp	-	-	S27	S27	-	-	-
strcoll	-	-	S27	S27	-	-	-
strcpy	S15	S22	S27	S27	-	-	-
strcspn	-	-	S27	S27	-	-	-
strftime	-	-	S1	-	S1	S1	-
strlen	S16	-	S27	-	-	-	-
strncat	S17	-	S27	S1	S21	-	-
strncmp	-	-	S1	S1	S21	-	-
strncpy	S17	-	S1	S1	S21	-	-
strpbrk	-	S2	S28	S27	-	-	-
strrchr	-	S2	S28	-	-	-	-
strspn	-	-	S27	S27	-	-	-
strstr	-	S2	S28	S27	-	-	-
strtod	-	-	S27	-	-	-	-
strtok	-	S2	S26	S27	-	-	-

Function	f(0)	f(r)	f(1)	f(2)	f(3)	f(4)	f(5)
strtol	-	-	S27	-	-	-	-
strtoul	-	-	S27	-	-	-	-
strxfrm	-	-	S27	S27	-	-	-
tmpfile	-	S2	-	-	-	-	-
ungetc	-	-	-	S1	-	-	-
vfprintf	-	-	S1	S27	S1	-	-
vprintf	-	-	S27	S1	-	-	-
vsprintf	-	-	S27	S27	S1	-	-
wcstombs	-	-	S1	S1	-	-	-
wctomb	-	-	S1	-	-	-	-

Semantics

S1 Checks the argument and if it is a pointer and can potentially be NULL issues a warning. The warning number depends on the likelihood that the pointer is NULL: **418** == certainty, **668** == possible, **802** == conceivable.

S2 Indicates that there is a possibility that the return pointer is NULL.

S3 (**abort** and **exit**) the function does not return.

S4 (**calloc**) the length of the returned buffer is the product of arguments 1 and 2.

S5 (**fgets**) complains if the 2nd argument (as a signed or unsigned integer) exceeds the data area represented by the 1st argument.

S6 (**fread**) complains if the product of the 2nd and 3rd arguments exceeds the buffer length implied by the 1st argument.

S7 (**free**) the argument is a pointer, which is subsequently regarded as uninitialized. Also the pointer may not be a kind derived either from **new** or **new[]**.

S8 (**fwrite**) complains if the product of the 2nd and 3rd arguments exceeds the buffer length implied by the 1st argument.

S9 (**malloc**, **operator new**, **operator new[]**) the length of the returned buffer is the value of the 1st argument.

S10 (**memchr**) if the 3rd argument is greater than the size of the buffer indicated by the 1st argument, a diagnostic (access beyond) is issued.

S11 (**memcmp**) if the 3rd argument is greater than the size of data area represented by either the 1st argument or the 2nd argument, a diagnostic is issued.

S12 (**memcpy**) a complaint is issued if the 3rd argument is greater than the size of the data area represented by either the 1st argument (overrun) or 2nd argument (access beyond).

S13 (**memset**) if the 3rd argument is greater than the size of the buffer indicated by the 1st argument, a diagnostic (data overrun) is issued.

S14 (**setjmp**) an informational message (**748**) is issued about the existence of register variables in any function calling the function. Also, registers are considered possibly uninitialized at that point.

S15 (**strcat** and **strcpy**) a diagnostic is issued if the size of the buffer area indicated by the 2nd argument is larger than the size of the buffer area indicated by the 1st argument.

S16 (**strlen**) the value returned by the function is assumed to be the size of the buffer (minus 1) represented by the 1st argument.

S17 (**strncpy** and **strncat**) issues a complaint (data overrun) if the value of the 3rd (integral) argument is greater than the size of the buffer represented by the 1st argument.

S18 (**gets**) issues a warning (**421**) at the very use of the function.

S19 (**fclose**) the argument is a pointer, which should be non NULL and is subsequently regarded as uninitialized.

S20 (**__assert**, **___assert**) the argument can be assumed to be true (i.e. non zero). See Section 10.2.1 The assert remedy

S21 Checks the presumed integral argument for negativity and if there is the possibility that the argument is negative will issue a warning (**422** == certainty, **671** == possible, **807** == conceivable).

S22 (**strcpy**) the return value is guaranteed to be not NULL.

S23 (**realloc**) the length of the return buffer is the value of the 2nd argument.

S24 (**realloc**) the 1st argument is thereafter considered as possibly uninitialized (unless of course it is immediately reassigned to the return value of **realloc**, which is the usual case).

S25 Checks the pointer argument for NULL (like **S1**) and designates that the type of the return value will be the type of the argument. This gets around a C problem where, for example, **strchr()** is typed:

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

If a **const** pointer is passed in as 1st argument then the return type for that call will be regarded as **const char ***. If it were regarded as a **char *** as the declaration says, then the **const** information will not be protected.

S26 The argument is considered a nul-terminated string.

S27 This combines semantics **S1** and **S26**. The pointer is checked for NULL (like **S1**) and the argument is considered a nul-terminated string.

S28 This combines semantics **S25** and **S26**. The argument is checked for NULL and the argument type is passed through (like **S25**) and the argument is considered a nul-terminated string.

S29 (**operator delete**) the argument is a pointer which is subsequently regarded as uninitialized. Also the pointer may not be a kind derived either from **malloc** or **new[]**

S30 (**operator delete[]**) the argument is a pointer which is subsequently regarded as uninitialized. Also the pointer may not be a kind derived either from **malloc** or **new**.

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 functionality, **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 ***ith 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 ***ith 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: **?:**

11.2.2 Semantic Expressions

Operators, parentheses and constants have their usual C/C++ meaning. Also the precedence of operators are identical to C/C++.

There may be at most two expressions in any one **-sem** option, one expressing Return semantics and one expressing Function-wide semantics.

Return semantics

An expression involving the return value (one of **@n**, **@p**, **@P**) is a Return semantic and indicates something about the returned value. For example, if the semantics for **strlen()** were given explicitly, they might be given as:

```
-sem( strlen, @n < 1p, 1p )
```

In words, the return value is strictly less than the size of the buffer given as first argument. Also the first argument should not be null.

To express further uncertainty about the return value, one or more expressions involving the return value may be alternated using the **||** operator. For example:

```
-sem( fgets, @p == 1p || @p == 0 )
```

represents a possible Return semantic for the built-in function **fgets**. Recall that the **fgets** function returns the address of the buffer passed as first argument unless an end of file (or error) occurs in which case the null pointer is returned. If the Return semantic indicates, in the case of a pointer, that the return value may possibly be zero (as in this example) this is taken as a possibility of returning the null pointer.

As another example:

```
-sem( lookup, 2n == LOCATE ? (@p==0||@p==1) : @p==1)
```

This is a Return semantic that says that if the 2nd argument of the function **lookup** is equal to **LOCATE** then the return pointer may or may not be null. Otherwise we may assume that the return value is a valid non-null pointer. This could be used as follows:

```
#define LOCATE 1
#define INSTALL 2
Symbol *lookup (const char *, int );
 ...
    sym = lookup("main", INSTALL);
    sym -> value = 0;   /* OK */
    sym = lookup("help", LOCATE);
    v = sym -> value;   /* warning - could be NULL */
```

Here the first return value from **lookup** is guaranteed to be non-null, whereas the second may be null or may not be.

We caution the reader that the following, apparently equivalent, semantic does not work.

```
-sem( lookup, @p == (2n != LOCATE) || @p == 1 )
```

The OR (**||**) is taken to mean that either side or both could be true with some probability but there is no certainty deduced that one or the other must be true.

Flagging the return value

Consider the example:

```
char *p, *q = 0;
p = malloc(10);
p = q;          // Warning -- memory leak
```

We are able to issue a Warning because the return from **malloc** has an allocation flag that indicated that the returned value points to a freshly allocated region that is not going to be freed by itself.

The seemingly equivalent semantic option:

```
-sem( my_alloc, @P == 1n )
```

associates no allocation flag with the returned pointer (only the size of the area in bytes).

To identify the kind of storage that a function may return, three flag-endowing functions have been added to the allowed expression syntax of the **-sem** option:

a region allocated by **malloc** to be released through **free**
a region allocated by **new** to be released through **delete**
a region allocated by **new[]** to be released through **delete[]**

In each case, the ***exp*** is the size of the area to be allocated. For example, to simulate **malloc** we may have:

```
-sem( my_alloc, @P == malloc(1n) )
```

By contrast the semantic:

```
-sem( some_alloc, @p == malloc(1n) )
```

indicates, because of the lower case '**p**', that the size of the allocated region is measured in allocation units. Thus the **malloc** here is taken to indicate the type of storage (freshly allocated that should be **free**) and not as a literal call to **malloc** to allocate so many bytes.

As another example:

```
-sem( newstr, @p == (1p ? new[](1p) : 0) )
```

In words, this says that **newstr** is a function whose return value will, if the first argument is a non-null pointer, be the equivalent of a **new[]** of that size. Otherwise a NULL will be returned.

Function-wide semantics

An expression that is not a Return semantic is a 'Function-wide' semantic (to use the terminology of Section 11.1.1 Special Functions). It indicates a predicate that should be true. If there is a decided possibility that it is false, a diagnostic is issued.

What constitutes a "decided possibility"? This is determined by considerations described in Section 10.2 Value Tracking. If nothing is known about a situation, no diagnostic is issued. If what we do know suggests the possibility of a violation of the Function-wide semantic, a diagnostic is issued.

For example, to check to see if the region of storage passed to function **g()** is at least 6 bytes you may use the following:

```
//lint -sem( g, 1P >= 6 ) 1st arg. must have at least 6 bytes
short a[3];      // a[] has 6 bytes
short *p = a+1;  // p points to 4 bytes
void g( short * );

g( a );          // OK
g( p );          // Warning
```

Several constraints may be AND'ed using the **&&** operator. For example, to check that **fread(** ***buffer*****,** ***size*****,** ***count*****,** ***stream*** **)** has non-zero second and third arguments and that their product exactly equals the size of the buffer you may use the following option.

```
-sem( fread, 1P==2n*3n && 2n>0 && 3n>0 )
```

Note that we rely on C's operator precedence to properly group operator arguments.

To continue with our example we should add Return Semantics. **fread** returns a value no greater than the third argument (***count***). Also, the first and fourth arguments should be checked for null. A complete semantic option for **fread** becomes:

```
-sem( fread, 1P==2n*3n && 2n>0 && 3n>0, @n<=3n, 1p, 4p )
```

It is possible to employ symbols in semantic expressions rather than hard numbers. For example:

```
//lint -sem( f, 1p > N )
#define N 100
int a[N]; int b[N+1];
void f( int * );
    ...
    f( a );   // warn
    f( b );   // OK
```

Just as is the case with **-function**, **-sem** may be applied to member functions. For example:

```
//lint -sem( X::cpy, 1P <= BUFLEN )

const int BUFLEN = 4;

class X
    {
  public:
    char buf[BUFLEN];
    void cpy( char * p )
        { strcpy( buf, p ); }
    void slen( char * p );
```

```
        };

    void f( X &x )
        {
        x.cpy( "abcd" );    // Warning
        x.cpy( "abc" );     // OK
        }
```

In this example, the argument to **X::cpy** must be less than or equal to **BUFLEN**. The byte requirements of "**abcd**" are 5 (including the nul character) and **BUFLEN** is defined to be 4. Hence a warning is issued here.

To specify semantics for template members, simply ignore the angle brackets in the name given to **-sem**. The semantics will apply to each template instantiation. For example, the user below wants to assign the custodial semantic to the first argument of the **push_back** function in every instantiation of template **list**. This will avoid a Warning **429** when the pointer is not deleted in **f()**.

```
//lint -sem( std::list::push_back, custodial(1) )

namespace std
    {
    template< class T >
        class list
            {
          public:
            void push_back( const int * );
            };
    }

std::list<int*> l;

void f()
    {
    int *p = new int;
    l.push_back( p );      // OK, push_back takes custody
    }
```

11.2.3 Notes on Semantic Specifications

1. Every function has, potentially, a Return semantic (**r**), a Function-wide semantic (**0**), flag semantics (**f**), and Argument semantics for each of the arguments and the implied **this** argument (**t**). An expression of the form ***i*p** when it stands alone and is not part of

another expression becomes an Argument semantic for argument ***i*** (presumably a pointer argument). Thus, for the option

```
-sem( f, 2p, 1p > 0 )
```

2p becomes an Argument semantic (the pointer should not be NULL) for argument 2. We can transfer this semantic to, say, the 3rd argument of function **g** by using the option

```
-function( f(2), g(3) )
```

The expression **1p>0** becomes the Function-wide semantic for function **f** and can be transferred via the 0 subscript as in:

```
-function( f(0), g(0) )
```

We could have placed these two together as one large semantic as in:

```
-sem( f, 2p && 1p > 0 )
```

The earlier rendition is preferred because there is a specialized set of warning messages for the argument semantic of passing null pointers to functions.

2. Please note that **r_null** and an expression involving argument @ are Return semantics. You cannot have both in one option. Thus you cannot have

```
-sem( f, r_null, @p = 1p )
```

It is easy to convert this into an acceptable semantic as follows:

```
-sem( f, @p == 0 || @p == 1p )
```

3. The notations for arguments and return values was not chosen capriciously. A notation such as **@n == 2n** may look strange at first but it was chosen so as not to conflict with user identifiers.

4. Please note that the types of arguments are signed integral values. Thus we may write

```
-sem( strlen, @n < 1p )
```

We are not comparing here integers with pointers. Rather we are comparing the number of items that a pointer points to (an integer) with an integral return value.

For uniformity, the arithmetic of semantics is signed integral arithmetic, usually long precision. This means that greater-than comparisons with numbers higher than the largest signed **long** will not work.

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 i**nt 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

13.1 Order of Evaluation

Expressions whose value depends on the order-of-evaluation are flagged with Warning **564** or **864**. This is a very famous (or infamous) problem with C/C++ but very few compilers will diagnose it. In general, the compiler is not obligated to evaluate expressions left-to-right or indeed in any particular order. For example,

```
n++ + n
```

is ambiguous; if the left hand side of the binary **+** operator is evaluated first, the expression will be one greater than if the right hand side is evaluated first. Some other, more common examples are:

```
a[i] = i++;
f( i++, n + i );
```

In the first case, it looks as though the increment should take place after computing the array index, but, if the right hand side of the assignment operator is evaluated before the left hand side, the increment is done before the index is computed. Although assignment looks as though it should imply an order of evaluation it does not. The second example is ambiguous because the order of evaluation of arguments to a function is not guaranteed. The only operators that imply an order of evaluation are Boolean AND (**&&**), Boolean OR (**||**), conditional evaluation (**?:**), and the comma operator (**,**). Hence:

```
if( (n = f()) && n > 10 ) ...
```

works as expected and you don't get a warning, whereas:

```
if( (n = f()) & n > 10 ) ...
```

will elicit a message.

In general, for every binary operator that does not have an implied order of evaluation, the set of variables modified or potentially modified by each side is compared with the set of variables accessed by the other side and an appropriate message is issued for each common member. Under the control of an option called the References - Modify flag (Flag **frm**) variables may be considered modified if they are passed to a function whose argument is a non-const reference or a non-const pointer.

Volatile variables and functions are subject to added scrutiny. See Section 13.5 volatile Checking

When an object is passed to a reference to a non-const or when a pointer to the object is passed to a pointer to a non-const then the object is potentially modified. If that object is used elsewhere in the same expression without an intervening sequence point, a potential order of evaluation problem occurs. For example:

```
int g( int );
int h( int & );
int f( int k )
    {
    return g(k) + h(k);   // Info 864
    }
```

Here the call to **h()** could occur either before or after the call to **g()** with presumably different results.

The same considerations apply to passing the address of a variable to a function.

```
int g( const int * );
int h( int * );
int f( int k )
    {
    int m = g( &k ) + g( &k );  // no warning
    return g(&m) + h(&m);       // Info 864
    }
```

Message **864** is of special interest to object programming. For example:

```
void f( int, int );
class X { public: int bump(); int k; };
...
X x;
f( x.bump(), x.bump() );  // Info 864
```

Member functions that are declared const can be used repeatedly in the same expression. For example:

```
class X { public: int bump(); int get() const; ... };
...
X x;
int n = x.bump() - x.get();     // Info 864
n = x.get() + x.get();          // no message
```

The message is also extended to cover member function calls where the object is pointed to by a variable. For example:

```
class X { public: int bump(); int get() const; ... };
...
X *p;
```

```
int n = p->bump() - p->get();      // Info 864
n = p->get() + p->get();           // no message
```

Message **591** can uncover seemingly innocent uses of global variables which are in fact dependent on the order of evaluation. For example, using 2 passes,

```
extern int n;
void f() { n++; }
int g() { f(); return 1; }
int h() { return n + g(); }      // Warning 591
```

In this example it makes a difference which side of the **+** operator is evaluated first. If the function **g()** were called first you get a result that is one greater then if variable **n** were evaluated first.

13.2 Format Checking

PC-lint/FlexeLint completely checks for **printf** and **scanf** (and family) format incompatibilities. For example,

```
printf( "%+c", ... )
```

will draw a warning (**566**) because the plus flag is only useful for numeric conversions. There are over a hundred such combinations that will draw this warning and compilers do not normally flag the inconsistencies. Other warnings that complain about bad formats are **557** and **567**. We follow the formatting rules established by ANSI/ISO C.

Perhaps more importantly we also flag arguments whose size is inconsistent with some format (Warnings **558**, **559**, **560** and **561**). Thus, with **%d** format, both integers and **unsigned int** are allowed but not **double** and not **long** if these are larger than **int**. Similarly, **scanf** type formats require that the arguments be pointers to objects of the appropriate size. If only the type of the argument (but not its size) is inconsistent with the format character, Warnings **626** or **627** will be given.

The **-printf** and **-scanf** options allow a user to specify functions that resemble a member of the **printf** or **scanf** family. Also **-printf_code** and **-scanf_code** can be used to describe non-standard **%** codes. See Section 5.7 Other Options

13.3 Indentation Checking

Indentation checking can be used to locate the origins of missing left and right braces. It can also locate potential problems in a syntactically correct program. For example, consider the code fragment:

```
if( … )
    if( … )
        statement
else statement
```

Apparently the programmer thought that the **else** associates with the first **if** whereas a compiler will, without complaint, associate the **else** with the second **if**. PC-lint/FlexeLint will signal that the **else** is negatively indented with respect to the second **if**.

There are three forms of messages; Informational **725** is issued in the case where there is no indentation (no positive indentation) when indentation is expected, Warning **525** is issued when a construct is indented less than (negatively indented from) a controlling clause, and **539** is issued when a statement that is not controlled by a controlling clause is nonetheless indented from it.

Of importance in indentation checking is the weight given to leading tabs in the input file. Leading tabs are by default regarded as 8 blanks but this can be overridden by the **-t#** option. For example **-t4** signifies that a tab is worth 4 blanks (see the **-t#** option in Section 5.7 Other Options).

Recognizing indentation aberrations comes dangerously close to advocating a particular indentation scheme; this we wish to avoid. For example, there are at least three main strategies for indentation illustrated by the following templates:

```
if( e ) {
    statements
}

if( e )
    {
    statements
    }

if( e )
{
    statements

}
```

Whereas the indentation methods appear to differ radically, the only real difference is in the way braces are handled. Statements are always indented positively from the controlling clause. For this reason PC-lint/FlexeLint makes what is called a ***strong*** check on statements requiring that they be indented (or else a **725** is issued) and only a ***weak*** check on braces requiring merely that they not be negatively indented (or else a **525** is issued).

case, and **default** undergo a weak check. This means, for example, that

```
switch()
    {
case 'a' :
    break;
default:
break;
    }
```

raises only the informational message (**725**) on the second **break** but no message appears with the **case** and **default** labels.

The **while** clause of a **do** ... **while(e);** compound undergoes a weak check with respect to the **do**, and an **else** clause undergoes a weak check with respect to its corresponding **if**.

An **else if()** construct on the same line establishes an indentation level equal to the location of the **else** not the **if**. This permits use of the form:

```
if()
    statement
else if()
    statement
else if()
    statement

    ...
else
    statement
```

Only statement beginnings are checked. Thus a comment can appear anywhere on a line and it will not be flagged. Also a long string (if it does not actually begin a statement) may appear anywhere on the line.

A label may appear anywhere unless the **+fil** flag is given (Section 5.5 Flag Options) in which case it undergoes a weak check.

Message **539** is issued if a statement that is not controlled by a loop is indented from it. Thus:

```
while ( n > 0 );
    n = f(n);
```

draws this complaint, as well it should. It appears to the casual reader that, because of the indentation, the assignment is under the control of the **while** clause whereas a closer inspection reveals that it is not.

13.4 const Checking

const is fully supported. We recommend that you incorporate the use of **const** in your programming style as there are several unexpected benefits from using this keyword. Consider the program fragment:

```
char *strcpy( char *, const char * );
const char c = 'a';
const char *p = &c;

void main()
    {
    char buf[100];

    c = 'b';
    *p = 'c';
    strcpy( p, buf );
    ...
```

This will draw four separate messages. Clearly **c** and ***p**, since they are **const** should not be modified (Error **111**). Also, passing **p** as a first argument to **strcpy** draws a warning (**605**) because of an increase in pointer capability. Finally, passing **buf** as the second argument draws a warning (**603**) because **buf** hadn't been initialized and a function expecting a pointer to a **const** value will not do the initialization.

If your compiler does not support **const** you may wish to use:

```
#ifdef _lint
#define CONST const
#else
#define CONST
#endif
```

at the head of your source code. Then use **CONST** rather than **const** throughout.

13.5 volatile Checking

volatile has only modest error checking properties. A warning is issued when a variable declared **volatile** is used twice in the same expression (order of evaluation problems) and declarations containing the keyword are checked for consistency. A warning is issued when pointers declared as pointing indirectly to **volatile** objects are used indirectly twice in the same expression and when functions declared to return **volatile** values are called twice in the same expression. (They receive Warning **564**). For example:

```
volatile char *p;
volatile char f();

n = (f() << 8) | f();       /* Warning 564 */
n = (*p << 8) | *p;         /* Warning 564 */
```

The reason the warning is given is that it is presumed that each access of a **volatile** object or function produces a potentially different value and that the order of evaluation cannot be guaranteed.

You may declare functions to be **volatile** if they have side effects such as returning the next character of an input stream or global string. A pointer may be pointing to a **volatile** object if it is used for memory mapped I/O. If your compiler doesn't support **volatile**, you may want to hide this from your compiler using the method described for **const** (See Section 13.4 const Checking).

Alternatively, to turn off **volatile** checking you may use the option:

```
-dvolatile=
```

13.6 Prototype Generation

The following option is useful for migrating from K&R C to ANSI/ISO C/C++.

-od[**s**][**i**][**f**][*width*]**(***filename***)**

outputs declarations to *filename* and is frequently employed to generate a set of prototypes for the functions defined within a module. If **+od** rather than **-od** is used, output is appended to *filename*. For example:

```
lint alpha.c -od(alpha.h)
```

produces a header file to be **#include'**d by routines that use the services provided by **alpha.c**

A prototype will be generated for functions defined "old-style" as in:

```
int f(x) int x; {return x;}
```

as well as for functions defined "new-style" as in

```
int f(int x) {return x;}
```

The same prototype, i.e.,

```
int f(int);
```

is generated in each case. If there is a clash between the declaration and the definition, the definition wins. This is to make it possible to regenerate header files. If the variable-arguments flag

```
+fva
```

has been set for the function when declared or defined, then no list of parameter types is generated. Thus

```
/*lint +fva  */
int f();
/*lint -fva  */

int f(int x) {return x;}
```

results in:

```
int f();
```

being generated. If a limit on the number of arguments to be checked is provided as in:

```
/*lint +fva1  */
int g();
/*lint -fva  */

int g(x,y) int x,y; {return x+y;}
```

then the output of the **-od** file will contain:

```
int g(x,...);
```

-odi (static functions)

Prototypes for **static** functions (i.e., functions with internal linkage) are not automatically generated with **-od**. This is consistent with the idea that prototype output is intended for inter-module communication. To get prototypes for **static** (Internal) functions, as well as external objects, use: **-odi(*filename*)**.

-odf (only functions)

If **f** is specified as in **-odf**, output is limited to functions (no data declarations).

`-ods` (structs)

Consistent with the idea that you are starting with a program that already is properly header'ed, so that different modules already "see" all the **`struct`**, **`union`** and **`enum`** that they need, we do not normally generate definitions of these objects as part of **`-od`**, If you want them, use **`-ods(`*`filename`*`)`**.

`-od`*width*

Prototypes are broken (with a new-line character) after spaces, commas and semicolons whenever the current width exceeds the specified ***`width`*** (default width is 66).

Precautions with Prototypes

Prototypes do not play just the passive role of enabling compilers and lint processors to more intelligently diagnose errors. They also play an active role in silently converting arguments. Message **`747`** will detect flagrant conversions. You may also want to detect subtle conversions by turning on Elective Notes **`917`** and **`918`** (with the options **`+e917 +e918`**).

typedef Types in Prototypes

It is possible to produce prototypes containing **`typedef`** names. See Section 9.9 Strong Types and Prototypes

13.7 Exact Parameter Matching

Note: This section deals only with old style (non-prototype) function calls.

Types of function parameters are not always taken literally. If a parameter is typed array, for example, this is considered a stylistic way of indicating pointer. With old-style function definitions, parameters of type **`char`**, **`unsigned char`**, **`short`**, **`unsigned short`**, and **`float`** are quietly promoted for the purpose of matching up with arguments. (However, for subsequent use within the function the original type is used). For example:

```
int f( ch, sh, fl, a )
    char ch;
    short sh;
    float fl;
    int a[10];
```

```
    {
...
```

is the start of an old-style function definition (i.e., a definition that does not place type information between the parentheses). For the purpose of detecting type conflict, PC-lint/FlexeLint will promote the types of **ch** and **sh** to **int**, **fl** to **double**, and **a** to pointer to **int**. If, for example, a **char** argument is passed as the first argument to **f()** this argument is also promoted by the rules of C to type **int** so that no mismatch is reported.

But it can be argued that some valuable type information is lost in this way. If an **int** is accidentally passed as first argument to **f()** the mismatch would go unreported. For this reason several flags are available to inhibit the usual promotion rules for parameters of this type:

+fxa eXact Array matching
+fxc eXact Char matching
+fxf eXact Float matching
+fxs eXact Short matching

These flags are effective only if the formal parameter is the one that is normally promoted. Consider

```
char ch;
    int g(i)
    int i;
    { ... }
... g( ch );
```

Here, the actual argument **ch** is considered matched against the formal parameter **i** even if the **+fxc** flag is set. On the other hand passing an **int** to the first argument of **f()** (previous example) would be flagged.

With exact array matching, for example, only an array of 10 **int**'s may be passed as fourth argument to function **f()** (previous example). Pointers may not be passed to array parameters but, as indicated above, array arguments may be passed to pointer parameters.

With **char** and **short** exact matching, the argument must be the exact type declared or a compatible constant. However, if the argument is an expression involving an operation other than the conditional (**?:**), the operation is assumed to be carried out with at least **int** precision. It will not match a **char** or **short** parameter.

To be compatible with a parameter typed **char** or **short**, constants need to be able to fit within the type without loss of precision. For example, **0** is compatible with **char** and **short** as well as with **int**.

When PC-lint/FlexeLint encounters calls to a function before seeing a definition (or a prototype) there is a slight problem. For example, if it sees:

```
f( 513, 'a' );
...
f( 'b', 814 );
```

Then what should PC-lint/FlexeLint record with regard to the arguments being passed to **f()**? Remember that no error should be reported if the parameters are subsequently discovered to be typed **int**. A worst case argument is saved, i.e., one that will match the fewest subsequent types. In this case it will be recorded that **f()** had been passed two **int**'s. If the definition:

```
void f( i, c )
        int i;
        char c;
        { ... }
```

is later discovered, then a mismatch is reported for the 2nd parameter. Unfortunately the position information of the offending argument will be lost because the information about the arguments had been derived from two different places. You will see "location unknown" in the message. If you can't find the position by just searching, reorder the modules so that the definition appears first. It may be more convenient to place a truncated definition in a dummy module before all the other modules. You'll get a duplicate definition, but who cares?

The **+fxc** and **+fxs** options may be useful if you are matching old-style function definitions with new-style prototypes. For example:

```
void g(char);

void g(c) char c; { ... }
```

is considered erroneous by PC-lint/FlexeLint since by the rules of ANSI/ISO the first **char** does not get promoted but the second one does. On the other hand, if the second **char** is changed to read **int** then the Microsoft compiler reports a type mismatch. Perhaps this is fair since ANSI/ISO C considers both sequences to be erroneous since a new-style prototype is being mixed with an old-style definition. A way to get around this difficulty and still retain the old-new confrontation is to use the **+fxc**.

Note: although lint does not complain about the argument difference it will complain because **g()** is retyped (type difference = promotion). To get lint to completely ignore this, it is necessary to also use the option: **-etd(promotion)**.

With **float** exact matching the considerations are similar to the case of **char** and **short** exact matching. Only constants that are **float** (such as **1.2f**) are considered compatible with a **float** parameter. The previously cited PC-lint/Microsoft conflict does not occur with the **float** type so that the use of **+fxf** may not be as compelling as with the other flags.

13.8 Weak Definials

The ***weak definials*** consist of the following:

macro definitions
typedef names
declarations
struct, **union** and **enum** definitions and members
template names

They are compile-time entities and for this reason, perhaps, they are not used as carefully or as scrupulously as run-time objects. Their definitions may be redundant or may lay around unused. Sometimes they are defined inconsistently across modules. Because they are only compile-time entities, they are referred to as 'weak'. The word 'definial' means simply that which is defined. It has the benefit of no prior use and hence semantic neutrality in C/C++. Where there is no possibility of confusion, we will use the word 'definial' as an abbreviation for the term 'weak definial'.

The weak definials are important because they represent those entities normally placed into header files to provide communication for the many modules that comprise a program. To determine whether a header file is unused or not depends upon whether any of its weak definials have been used.

Informational messages in the range **749**-**769** and **1749**-**1769** are reserved for the weak definials. PC-lint/FlexeLint is able to report on unused header files (**766**), definials within (non-library) header files that are not used (**755**-**758**, **768**, **1755**), non-header definials that are not used (**750**-**753**, **1750**), redundant definials (**760**-**763**) and conflicting definials.

If you run lint on some previously unlinted source code, you may well want to turn these messages off. However, if you want to see just header anomalies, you might try:

```
lint -w1 +e749 +e?75? +e?76? ...
```

13.8.1 Unused Headers

Whether a header file is used or not depends on whether any of its definials have been used by any file other than itself or the set of files in the same ***group***. A group of headers is defined below. For now assume a group contains just the file itself. For example, let the complete contents of **hdr.h** be:

```
typedef int INT;
extern INT f();
```

Assume a single module includes this header file but makes no use of either **f** or of **INT**. The definial **INT** would be considered used by virtue of its appearance within the declaration of **f** and **f** would be reported unused. The header file would be reported unused by the module because the only use of any of its definials was a self reference, a reference to **INT** from within the same header file. If the declaration of **f** were removed, then **INT** would be reported as unused and **hdr.h** would also be reported as unused.

Consider the following example:

```
hdr.h:

        typedef int INT;


alpha.c:

        #include "hdr.h"
        typedef int INT;
        INT x = 0;
```

Is the definial **INT** within **hdr.h** being used or not? Is the header file **hdr.h** being used? Since we have two identical declarations for **INT** it is hard to say. What we do in this case is report that the second **typedef** is redundant. We then act as if the second never appeared and so the header file appears to have been used. If the second **typedef** were a different type, an error would be reported, and the first **typedef** would be considered unused.

A *group* of headers is defined by a **#include** at the module level and contains the included header and all other headers directly or indirectly included by that header. It is important to note that the base of a group is a header directly included by a module.

Consider an example where **module.c** includes **group.h**, which, in turn, includes **hdr1.h** and **hdr2.h**. If **hdr2.h** uses something out of **hdr1.h** this is not considered a module use since both headers lie in the same group (consisting of **group.h**, **hdr1.h** and **hdr2.h**). If **module.c** makes no other reference to any item in the group you will receive a message (**766**) header file not used for **group.h** (i.e. the base of the group.) You will not normally receive a message about the subheaders **hdr1.h** and **hdr2.h** not used (this is available as Elective Note **966**). Experience has shown that programmers are much more interested in eliminating **#include** statements at the module level. Modifying headers themselves may be unwise owing to the variety of contexts in which headers are used.

In a similar vein, if **module.c** uses something from, say **hdr1.h**, but nothing from **group.h**, **group.h** is nonetheless considered used and **766** is not issued. You can learn that **group.h** was not used directly by enabling Elective Note **964** but this fact is not normally very interesting as it would require a rearrangement of header information to exploit.

If **module.c** includes another header **group2.h**, then any reference to an element of **group.h** by this new header, lying as it does in some other group, is considered a reference by the module and serves to suppress **766** for **group.h**.

13.8.2 Removable from Header

A special message (**759**) is issued for objects declared in headers but then not referenced outside the module that defines them. (This message is automatically suppressed if there is only one module being processed). If a declaration is used by only one module, it can be removed from the header file thereby reducing its size. Header files have a tendency to become big and fat; compilers are always indicating when something has to be added but hardly ever indicate when something can be deleted; this produces uni-directional growth. Message **759** is intended to combat this tendency.

13.8.3 static-able

A related message (**765**) is the identification of all objects that are 'file-scopable'; i.e., external objects that may be tagged **static** and hence not placed into the pool of external names. A programmer may not at all be interested in making everything **static** because modern debuggers sometimes depend critically on such external symbols. However you may wish to employ the following technique:

```
#if debug && !defined(_lint)
#define LOCALF
#define LOCALD extern
#else
#define LOCALF static
#define LOCALD static
#endif
```

LOCALD stands for local Declaration.
LOCALF stands for local deFinition.

These are used as:

```
LOCALD double func();
...
LOCALF double func() {return 37.5; }
```

For debugging (and provided we are not running PC-lint/FlexeLint) the function **func** is external and its name is available to the debugger. Otherwise it is made **static**. The macros provide good documentation and PC-lint/FlexeLint enforces compliance.

13.9 Unix Lint Options

The following options are available for compatibility with other static analyzers, in particular with the original Unix ***lint***. They may be embedded within C/C++ source code, where they appear as a comment. They are all of the form:

```
/* Optional-blanks  Keyword  Optional-blanks */
```

- **/* LINTLIBRARY */**

  This is equivalent to **/*lint -library */**. See Section 6.2 Library Modules

- **/* ARGSUSED */**

  Inhibits complaints about function parameters not being used for the duration of a single function. It is placed just before a function definition and is equivalent to:

  ```
  -efunc(715, function-name)
  ```

- **/* VARARGS[*N*] */**

  When this option is placed before a function declaration (or definition) it has the effect of **/*lint +fva[*N*] */** for just one function. Like **ARGSUSED**, it has an automatic reset feature.

- **/* NOTREACHED */**

  This option is equivalent to **/*lint -unreachable */**

- **/* NOSTRICT */**

  This inhibits certain kinds of strict type-checking for the next expression. The type differences that are relaxed are those denoted as **nominal**, **signed/unsigned**, **ellipsis**, **promotion** and **ptrs to incompatible types**.

  The equivalent PC-lint/FlexeLint option is:

  ```
  /*lint -save -etd(nominal,signed/unsigned,ellipsis)
       -etd(promotion,"ptrs to incompatible types") */
  ... expression ...
  /*lint  -restore  */
  ```

  (See ***TypeDiff*** in Chapter 19. MESSAGES.)

- `/* FALLTHROUGH */`
- `/* FALLTHRU */`

  These options are equivalent to `-fallthrough`.

```
void f( int i, const char * s, ... )
       {
         switch( i )
              {
         case 1:
              i = 2;
              /*FALLTHRU*/  // same as /*lint -fallthrough */
         case 2:
         default:
              break;
              }
         // ...
         }
```

  See `-printf` in Section **5.7 Other Options**

- `/* PRINTFLIKE2 */`

  This option when placed just before a function is equivalent to our `-printf(2,f)`. See `-printf` in Section **5.7 Other Options**

13.10 Static Initialization

Traditional lint processors do not flag uninitialized static (or global) variables because the C/C++ language defines them to be `0` if no explicit initialization is given. But uninitialized statics, because they can cover such a large scope, can be easily overlooked and can be a serious source of error. PC-lint/FlexeLint will flag static variables that have no initializer and that are assigned no value. For example, consider:

```
int n;
int m = 0;
```

There is no real difference between the declarations as far as C/C++ is concerned but PC-lint/ FlexeLint regards `m` as being initialized and `n` not initialized. If `n` is nowhere assigned a value, a complaint will be emitted.

13.11 Size of Scalars

Since the user of PC-lint/FlexeLint has the ability to set the sizes of various data objects (See the **-s**.. options in Section 5.3 Size and Alignment Options), the reader may wonder what the effect would be of using various sizes.

Several of the loss of precision messages (**712**, **734**, **735** and **736**) depend on a knowledge of scalar sizes. The options **-ean** and **-epn** only suppress **long**/**int**/**short** mismatches if they are the same size. Similarly, **-eas** and **-eps** depend on the sizes of data items. The legitimacy of bit field sizes depends on the size of an **int**. Warnings of format irregularities are based in part on the sizes of the items passed as arguments.

One of the more important effects of type sizes is the determination of the type of an expression. The types of integral constants depend upon the size of **int** and **long** in ways that may not be obvious. For example, even where **int** are represented in 16 bits the quantity:

```
35000
```

is **long** and hence occupies 4 (8-bit) bytes whereas if **int** are 32 bits the quantity is a four byte **int**. If you want it to be **unsigned** use the **u** suffix as in **35000u** or use a cast.

Here are the rules: (these ANSI rules may be partially suppressed with the **+fis** flag) the type of a decimal constant is the first type in the list (**int**, **long**, **unsigned long**) that can represent the value. The maximum values for these types are taken to be:

Largest is 1 less than 2 raised to the power:

```
int             sizeof(int)*bits-per-byte - 1
unsigned        sizeof(int)*bits-per-byte
long            sizeof(long)*bits-per-byte - 1
unsigned long   sizeof(long)*bits-per-byte
```

The quantities **sizeof(int)** and **sizeof(long)** are based on the **-si#** and **-sl#** options respectively.

The type of a hex or octal constant, however, is the first type on the list (**int, unsigned int, long, unsigned long**). For any constant (decimal, hex or octal) if it has a **u** suffix, one selects from the list (**unsigned int, unsigned long**). If an **L** suffix, the list is (**long, unsigned long**). If both suffixes are used then the type must be **unsigned long**.

The size of scalars enters into the typing of intermediate expressions in a computation. Following ANSI/ISO standards, PC-lint/FlexeLint uses the so-called *value-preserving* rule for promoting types. Types are promoted when a binary operator is presented with two unlike types and when unprototyped function definitions specify subinteger parameters. For example, if an **int** is added to an **unsigned short**, then the latter is converted to **int** provided that an **int** can hold all

values of an **unsigned short**; otherwise, they are both converted to **unsigned int**. Thus the signedness of an expression can depend on the size of the basic data objects.

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.

14. NON-STANDARD EXTENSIONS

This chapter describes generally accepted, non-standard extensions to the C/C++ language, which have been optionally incorporated into PC-lint/FlexeLint. Generally, these features reflect current practice in the MS-DOS world and may be disallowed by using the **-A** flag. Non-standard preprocessor extensions are covered in Section 15.4 Non-Standard Preprocessing.

14.1 Memory Models

Memory models have been introduced into a number of C/C++ compilers to support the Intel 8086 through 80286 chips and, in some cases, the 80386 and 80486 chip. If you are not concerned with the segmented architecture of these chips you can ignore this section.

There are four distinct memory models and these can be selected by one of the **-m**... options described in Section 5.7 Other Options.

option	*model name*	*data pointers*	*program pointers*	
-mS	small	**near**	**near**	(Default)
-mD	large data (compact)	**far**	**near**	
-mP	large program (medium)	**near**	**far**	
-mL	large	**far**	**far**	

In addition to selecting a memory model, it is possible for a programmer to override the default for any particular pointer. For example:

```
char far *p;
```

indicates that **p** is a pointer to a **far char** and is hence a **far** pointer. In a similar way, pointers can be declared to be **near** and **huge** (**huge** is taken as a synonym for **far**). Data objects and functions can also be declared as having the property of **near** or **far** and pointers to such objects automatically become **near** or **far** as appropriate.

It suffices to say that PC-lint/FlexeLint supports the Microsoft conventions for the use of these keywords and that these can be enabled (if they are not pre-enabled in your implementation) by selecting the option **+rw(*ms)**. This requests all the Microsoft reserved words. You may prefer to turn on just one or two of these reserved words. For example: **+rw(near,far)** enables just **near** and **far**.

It is also possible to disable these reserved words. by using the option **-rw(*ms)**. The **-A** option serves to flag such constructs. See Section 5.7 Other Options

Your version of PC-lint/FlexeLint is configured to have the system default sizes for **near** and **far** pointers to program and data. For cross-linting these can be set explicitly using a variation of the **-sp**... option as previously discussed in Section 5.3 Size and Alignment Options.

-spN# indicates that the size of **near** pointers (both of program and data) is # bytes.

-spF# indicates that the size of **far** pointers (both program and data) is # bytes.

-spFD# indicates the size of a **far** data pointer.

-spFP# indicates the size of a **far** program pointer.

-spND# indicates the size of a **near** data pointer.

-spNP# indicates the size of a **near** program pointer.

-smpFP# indicates the size of a member **far** Program pointer.

-smpNP# indicates the size of a member **near** Program pointer.

14.2 Additional Reserved Words

Many compilers offer additional reserved words over those specified by ANSI/ISO. See Section 5.8 Compiler Adaptation for ways of coping with them.

15. PREPROCESSOR

15.1 Preprocessor Symbols

_lint The special preprocessor symbol **_lint** is pre-defined with a value representing the version and patch level of PC-lint/FlexeLint. The primary purpose of this symbol is to enable the programmer to determine whether PC-lint/FlexeLint is processing the file.

For example, if you have a section of code that is unacceptable to PC-lint/FlexeLint for some reason (such as in-line assembly code), you can use **_lint** to make sure that PC-lint/FlexeLint doesn't see it. Thus,

```
#ifndef _lint
...
Unacceptable coding sequence
...
#endif
```

will cause PC-lint/FlexeLint to skip over the elided material.

The value of **_lint** is 100 * Version Number + the number of patch.

Version	*Value of* **_lint**
9.00	900
9.00a	901
...	...
9.00z	926
9.00aa	927
10.00	1000

E.g.

```
#if _lint >= 900
    // use Version 9 feature
#endif

#if _lint != 902
    // not for Version 9.02
#endif
```

__cplusplus This symbol is defined (as "1") for each module that is interpreted as being a C++ module and is otherwise undefined. C++ modules are determined by extension and possibly by option. See Chapter 4. THE COMMAND LINE.

The following pre-defined identifiers begin and end with double underscore and are ANSI/ISO compatible.

__TIME__ The current time
__DATE__ The current date
__FILE__ The current file
__LINE__ The current line number
__STDC__ Defined to be 1.
__STDC__VERSION__ This is undefined for C++. It is defined for C by default as '**199901L**'. If you select an earlier version of C using **-A** option as in **-A(C90)** this will be undefined.
__STDC__HOSTED__ This is defined whenever **__STDC_VERSION__** is defined. When defined it is defined to be 0.

Compiler-dependent preprocessor symbols may also be established as described in Section 5.8 Compiler Adaptation.

15.2 include Processing

When a **#include "*filename*"** directive is encountered

1. there is first an attempt to **fopen** the named file. But what is the named file? If the **fdi** flag is OFF the name between quotes is used. If the **fdi** flag is ON, the name of the including file is examined to determine the directory. This directory is prefixed to ***filename***. The directory of the including file is found by scanning backward for one of possibly several system-related special characters. If the **fopen** fails, we go to step 2.

2. there is an attempt to prepend (in turn) each of the directories associated with options of the form:

   -i*directory*

   in the order in which the options were presented. If this fails we go to step 3.

3. On systems supporting environment variables, each directory in the sequence of directives specified by the **INCLUDE** environment variable is prepended to the file.

4. There is an attempt to **fopen** the file by the name provided, without considering flag **fdi**.

If the include directive is of the form

#include <*filename*>

then the processing is the same except that step 1 is bypassed.

15.2.1 INCLUDE Environment Variable

The **INCLUDE** environment variable may specify a search path in which to search for header files (**#include** files). For example:

```
set INCLUDE=b:\include;d:\extra
```

specifies that, should the search for a **#include** file within the current directory fail, a search will be made in the directory **b:\include** and, on failing that, a search will be made in the directory **d:\extra**. This searching is done for modules as well as **#include** files. You may select an environment variable other than **INCLUDE**. See the **-incvar** option in Section 5.7 Other Options.

Notes:

1. No blank may appear between '**INCLUDE**' and '**=**'. Blanks adjacent to semicolons (**;**) are ignored. All other blanks are significant
2. A terminating semi-colon is ignored.
3. This facility is in addition to the **-i**... option and is provided for compatibility with a number of compilers in the MS-DOS environment.
4. Any directory specified by a **-i** directive takes precedence over the directories specified via the **INCLUDE** environment variable.

15.3 ANSI/ISO Preprocessor Facilities

ANSI/ISO preprocessing is assumed throughout. If the K&R preprocessor flag is set (**+fkp**) the use of ANSI/ISO (over K&R) is flagged.

15.3.1 #line and

A C/C++ preprocessor may place **#line** directives within C/C++ source code so that compilers (and other static analyzers such as PC-lint/FlexeLint) can know the original file and original line numbers that produced the text actually being read. In this way, these processors can report errors in terms of the original file rather than in terms of the intermediate text.

By default, **#line** directives are processed. To ignore **#line** directives use the option **-fln** (See Section 5.5 Flag Options). Some systems support # as an abbreviation for **#line**. To enable # use **++fln** (increment the **fln** flag to 2). For example,

```
//lint ++fln allow # as abbreviation for #line.
#line 32 “alpha.cpp”
    n = m/0;   // reported as a problem with line 32 of alpha.cpp
```

```
#  20  “alpha.cpp”
    x = 1;
    y = x/0;   // reported as a problem with line 21 of alpha.cpp
```

15.4 Non-Standard Preprocessing

Preprocessor commands in this section need to be activated via the **+ppw** option. Also, their semantics may be copied via the **ppw_asgn** option. See Section 5.7 Other Options

15.4.1 #assert

#assert is supported to conform with Unix V Release 4. Thus

```
#assert predicate(token-sequence)
```

will assume the truth of the ***predicate*** when tested against the indicated ***token-sequence*** in a preprocessor conditional. Without the parenthetical expression, ***predicate*** is established to exist. For example,

```
#assert machine(pdp11)
```

makes

```
#if #machine(pdp11)
```

true.

A **#unassert** preprocessor directive with the same syntax as **#assert** undoes the effects of **#assert** and is compatible with the Unix convention. See also option **-a#...** in Section 5.8.3 Customization Facilities.

15.4.2 #c_include

Some compilers support a conditional include facility using the **#c_include** preprocessor directive. For example:

```
#c_include "filename"
```

(Angle brackets may be used instead of quotes to delimit the filename.) Conditional include means that a file is included if it has not already been included while processing a particular module.

To activate this preprocessor directive use the **ppw** option as follows:

```
+ppw(c_include)
```

15.4.3 #asm

#asm introduces assembly language constructs, which PC-lint/FlexeLint will not examine but merely skip over. The assembly language constructs will be terminated either by a single # (on a line by itself) or by **#endasm** (if enabled) or the equivalent.

#asm needs to be enabled with **+ppw(asm)**. For other ways of introducing assembly language see Section 5.8.6 In-line assembly code.

15.4.4 #dictionary

#dictionary is intended for VAX VMS users to mimic this preprocessor convention of the DEC VMS compiler. This is useful only for FlexeLint users. When a

```
#dictionary filename
```

is encountered **i_open()** (within file **custom.c**) is called with a special 2nd argument (equal to '#'). The user can then encode special actions required by the semantics of **#dictionary**. The **#dictionary** directive need not be specially enabled for VAX VMS, but for every other system a **+ppw(dictionary)** would be required before use.

15.4.5 #endasm

#endasm is the complement to **#asm**. It needs to be enabled before use. A typical enabling sequence is

```
+ppw( asm, endasm )
```

A typical use is:

```
#asm        /* begin assembly code */
MOV AH,O
ADD AH,AL
#endasm     /* end assembly code */
```

15.4.6 #import

This preprocessor directive is intended to support the Microsoft preprocessor directive of the same name. For example:

```
#import "c:\compiler\bin\x.lib"
```

will determine the base name (in this case "**x**") and attempt to include, as a header file, ***basename*.tlh**. Thus, for linting purposes, this directive is equivalent to:

```
#include "x.tlh"
```

Options that may accompany **#import** are ignored. When the (Microsoft) compiler encounters a **#import** directive it will generate an appropriate **.tlh** file if a current one does not already exist. PC-lint/FlexeLint will not generate this file.

When compiling, it is possible to place the generated **.tlh** file in a directory other than the directory of the importing file. If this option is chosen, then when linting, this other directory needs to be identified with a **-i** option or equivalent.

This preprocessor word is not enabled by default. It can be enabled via the **+ppw(import)** option. This option has been placed into the various compiler options files for the Microsoft C/C++ compiler.

15.4.7 #unassert

The **#unassert** preprocessor directive has the same syntax as **#assert** and undoes the effects of **#assert**. This is compatible with the Unix convention.

15.4.8 #include_next

#include_next is supported for compatibility with the GNU C/C++ compiler. It uses the same arguments as **#include** but starts the header file search in the directory just after the directory (in search order sequence) in which the including file was found. See Section 15.2 include Processing for a specification of the search order.

For example; suppose you place a file called **stdio.h** in a directory that is searched before the compiler's directory. Thus you could intercept the **#include** of **stdio.h** and effectively augment its contents as follows:

```
   stdio.h:

#include_next <stdio.h>
    ... augmentation
```

15.5 User-Defined Keywords

PC-lint/FlexeLint might stumble over strange preprocessor commands that your compiler happens to support. For example, some Unix system compilers support **`#ident`**. Since this is something that can NOT be handled by a suitable **`#define`** of some identifier we have added the **`+ppw(`*`command-name`*`)`** option ('**`ppw`**' is an abbreviation for PreProcessor Word). For example, **`+ppw(ident)`** will add the preprocessor command alluded to above. PC-lint/FlexeLint recognizes and ignores the construct.

16. LIVING WITH LINT

(or Don't Kill the Messenger)

The comments in this chapter are suggestive and subjective. They are the thoughts and opinions of only one person and for this reason are written in the first person.

When you first apply PC-lint/FlexeLint against a large C or C++ program that has not previously been linted, you will no doubt receive many more messages than you bargained for. You will perhaps feel as I felt when I first ran a Lint against a program of my own and saw how it rejected 'perfectly good' C code; I felt I wanted to write in C, not in Lint.

Stories of Lint's effectiveness, however, are legendary. PC-lint was, of course, passed through itself and a number of subtle errors were revealed (and continue to be revealed) in spite of exhaustive prior testing. I tested a public domain grep that I never dared use because it would mysteriously bomb. PC-lint found the problem -- an uninitialized pointer.

It is not only necessary to test a program once but it should be continuously tested throughout a development/maintenance effort. Early in Lint's development we spent a considerable effort, over several days, trying to track down a bug that Lint would have detected easily. We learned our lesson and were never again tempted to debug code before linting.

But what do you do about the mountain of messages? Separating wheat from chaff can be odious especially if done on a continuing basis. The best thing to do is to adopt a policy (a policy that initially might be quite liberal) of what messages you're happy to live without. For example, you can inhibit all informational messages by the option **`-w2`**. Then work to correct only the errors associated with the messages that remain. DO NOT simply suppress all warnings with something like: **`-e*`** or **`-w0`** as this can disguise hard errors and make subsequent diagnosis very difficult.

The policy will be automatically imposed by incorporating the error suppression options in a **`.lnt`** file (examples shown below) and it can gradually be strengthened as time or experience dictate.

Experience has shown that linting at full strength is best applied to new programs or new subroutines for old programs. The reasons for this is that the various decisions that a programmer has made are still fresh in mind and there is less hesitancy to change since there has been much less 'debugging investment' in the current design. Decisions such as, for example, which objects should be **`signed`** and which **`unsigned`**, can benefit from checking at full strength. Full strength can even mean torture testing. See Section 18.14 Torture Testing Your Code.

16.1 An Example of a Policy

An example of a set of practices with which I myself can comfortably live, is as follows.

Mistaking assignment for equality is a potential problem for C/C++. If a Boolean test is made of assignment as in

```
if( a = f( ) ) ...
```

PC-lint/FlexeLint will complain with message **720**. If the assignment is wrapped with parentheses as in

```
if( (a = f()) ) ...
```

A different message (**820**) is used. Combining the assignment with testing is such a useful operation that I'm happy to put up with an extra pair of parentheses. Therefore, I suppress **820** with the option

```
-e820
```

At one time I mixed **unsigned** and **signed** quantities with almost reckless abandon. I now have considerably more respect for the subtle nuances of these two flavors of integer and now follow a more cautious approach. I had previously employed the options

```
-e713  -e737  -eau
```

(**713** involves assigning unsigned to signed, **737** is loss of sign, and **eau** suppresses messages based on the fact that an argument and a parameter disagree in that one is **signed** and the other is **unsigned**). These inhibitions affect only some variation of assignment. We retain warnings about mixing **signed/unsigned** with binary operators.

I also no longer think it is a great idea to automatically inhibit **734** (sub-integer loss of precision). This message can catch all sorts of things such as assigning **int** to **short** when **int** is larger than **short**, assigning oversize **int** to **char**, assigning too large quantities into bit fields, etc.

I suppress messages about shifting **int** (and **long**) left but I want to be notified when they are shifted right as this can be machine-dependent and is generally regarded as a useless and hazardous activity. Therefore, I use **-e701 -e703**.

I routinely employ functions without a prior declaration allowing them to default to **int**. Therefore I use option **-e718** (function not declared).

I want to run my code through at least two passes so that cross-functional checks can be made. The option is

```
-passes(2)
```

I place my list of favorite error-suppression options in a file called **options.lnt**. It looks like this:

```
options.lnt:

-e720 or -e820  // test of assignment
-e701 -e703     // shifting int left is OK
-e718           // undeclared function
-e746           // allow calls w/o prototypes
-passes(2)      // use two passes
```

16.2 Recommended Setup

For PC-lint the **SETUP** program creates a batch file **lin.bat** that indirectly includes the3 centralized **options.lnt** file described in the previous section. This situation is depicted in *Figure* **Setup** below. The batch file **lin.bat**, needs to be placed in your path. When **lin.bat** is run from most directories, the file **std.lnt** is found in the PC-lint directory (owing to the **-i** option appearing in the batch file). This in turn includes a compiler options file and a centralized **options.lnt** as shown. When **lin.bat** is run from the directory **c:\alpha**, however, a local **std.lnt** (also shown in the figure) overrides the standard one. This local **std.lnt** establishes a memory model appropriate to the application being developed within the directory and adds other locally pertinent options as well.

If you switch back and forth between compilers you may wish to place the name of a generic compiler options file, such as **c.lnt**, within the centralized **std.lnt**.

Then copy a compiler-specific options file into your compiler's include directory under this generic name. For example:

```
copy co-aztec.lnt  c:\aztec\include\c.lnt
```

Thus, when you switch to the Aztec compiler you presumably change the **INCLUDE** environment variable so the appropriate **c.lnt** is found.

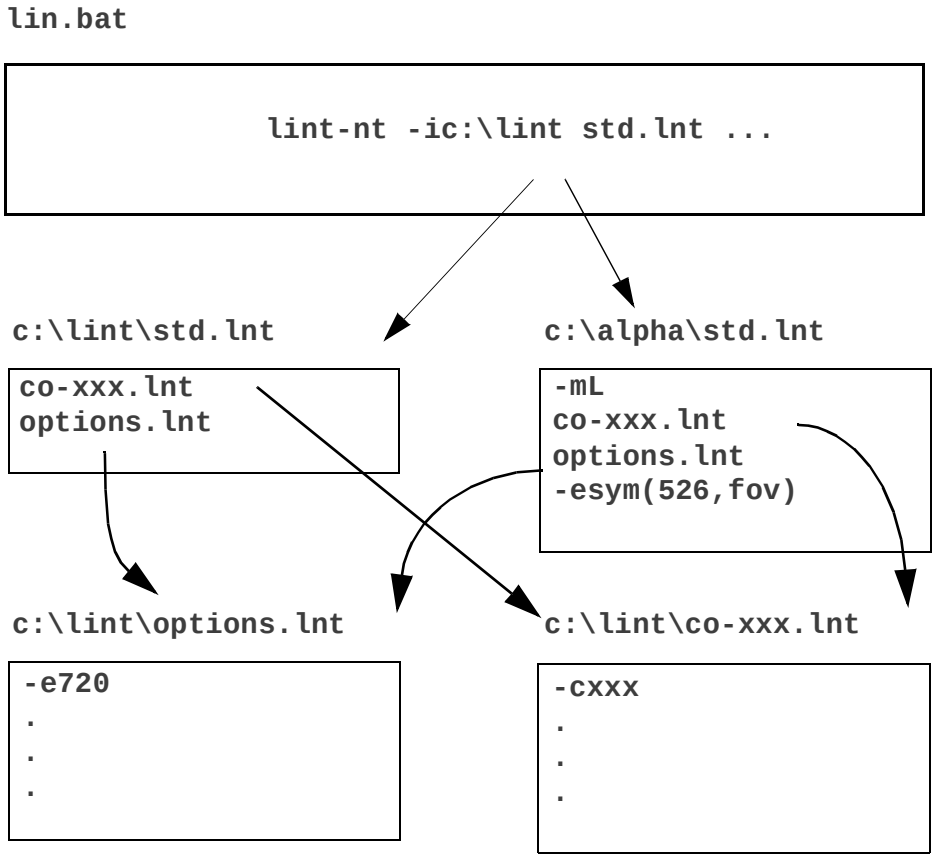


Figure **Setup**

Then copy a compiler-specific options file into your compiler's include directory under this generic name. For example:

```
copy co-aztec.lnt  c:\aztec\include\c.lnt
```

Thus, when you switch to the Aztec compiler you presumably change the **INCLUDE** environment variable so the appropriate **c.lnt** is found.

16.3 Using Lint Object Modules

For large projects (more than 50 source modules), I use Lint Object Modules (See Section 8.1 What is a LOB?).

For FlexeLint my **make** file setup is similar to that described in Section 8.5 Make Files.

For PC-lint, a typical **make** file has the form:

```
makefile:

  .c.lob:
      lint -u make.lnt $* -oo

  m1.lob:  m1.c

  m2.lob:  m2.c

  m3.lob:  m3.c

  m4.lob:  m4.c

  m5.lob:  m5.c

  project.lob:  m1.lob  m2.lob ... m5.lob
         lint make.lnt *.lob
```

In the above, `make.lnt` contains those things that the batch file `lin.bat` (described earlier) provided. In particular it contains:

```
make.lnt:

  -ic:\lint\
   std.lnt
  -os(temp)
  +v
```

16.4 Summarizing

In summary, establish procedures whereby PC-lint/FlexeLint may be conveniently accessed for a variety of purposes. Lint small pieces of a project before doing the whole thing. Establish an error-message suppression policy that may initially be somewhat relaxed and can be strengthened in time. Lint full strength on new projects.

But don't kill the messenger!

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 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.

17.2 Output Format Strings

The output format for each file may be specified by a format option in additional sub-options to the **program_info** option. Each format option will take the form:

```
format_category="String"
```

where ***String*** may contain plain characters, escape sequences and data field specifiers. You may specify the following formats: **format_file, format_type, format_symbol** and **format_macro**. The supported escape sequences are as follows:

```
\q (double quote)
\\ (backslash)
\a (alert)
\b (backspace)
\f (form feed)
\n (new line)
\t (horizontal tab)
\s (space)
\xdd (non-zero hexadecimal escape sequence)
%% (literal percent character)
```

Data field specifiers have the form:

```
%[FIELD]
```

where ***FIELD*** (note: UPPER case) is one of the data fields specified in **Section 17.1**. Some of the features of conversion specifiers usable with **fprintf()** may be used here. In particular, you may use the **-** flag to indicate left justification, and you may specify a minimum width and/or a precision (that is, a maximum width). E.g., if you want symbol names to be left-justified with a minimum width of 20 and a maximum width of 30, use:

```
+program_info( output_prefix="",
format_symbol="%-20.30[NAME] ... other fields ... \n"
)
```

Note that if the text of your format specifier contains a newline character as in:

```
+program_info( output_prefix="",
format_symbol="%[TYPE]
%[NAME]"
)
```

then the newline character will be converted to a single space. If you want a newline in your output, use **\n**.

By default, the formats are set as follows:

```
format_file="%[INDEX]\t%[NAME]\n"

format_type="%[INDEX]\t%[NAME]\t%[KIND]\n"

format_symbol="%[FILE]\t%[LINE]\t\"%[NAME]\"\t%[LINKAGE]\t
```

```
                %[SCOPE]\t%[KIND]\t%[TYPE]\n"

format_macro="%[FILE]\t%[LINE]\t\"%[NAME]\"\t%[PARAMETER_COUNT]\n"
```

17.3 Enabling and Suppressing Output

It is possible to establish format options and an output prefix without actually triggering the output of data. To do so, use the **program_info** option with the '**-**' prefix (instead of '**+**'). Output can then be triggered with **+program_info**. E.g.: Within **std.lnt**:

```
-program_info( output_prefix= )
```

On the command line:

```
lint std.lnt +program_info
```

If you do not want output for a particular category of program information, you may give an empty format specifier for it. E.g.:

```
+program_info( output_prefix="foo/",
               format_macro=,
               format_file= )
```

suppresses the output of the files **macro.txt** and **file.txt**.

By default, each file will begin with a single-line heading that indicates the format specification used therein. Output of this line may be suppressed by placing a **-** before the sub-option name. E.g.:

```
-program_info( output_prefix= )
-program_info( -format_symbol )
-program_info( -format_macro=%[NAME]\n )
+program_info
```

(Note that sub-options aggregate over invocations of the **-program_info** option.) This sequence of options instructs Lint to use no prefix in output filenames, to use the default format for symbols, to use a specified format for macros, and prevents the output of heading information in both of the resulting files. For files and types, the default format will be used and the heading will appear.

17.4 Flag fields

In some categories, records have a set of ***flag*** fields that indicate miscellaneous boolean states. Example: in C++, if the definition of a member function appears lexically in the class of which it is a member -- e.g.:

```
class A { void f() {} void g(); };
void A::g(){}
```

Then our symbol record for **A::f()** will have the flag **defined_in_class**, but for **A::g()**, that flag will be unset.

A possible format for symbols is:

```
+program_info( output_prefix=,
format_symbol="%[NAME], flags: '%[defined_in_class]' \n"
)
```

With this format and the above code sample, our **symbol.txt** would include the following two records (among others):

```
A::f, flags: 'defined_in_class'
A::g, flags: ''
```

17.4.1 File Flags

dsp -- A DSP file
cpp -- A C++ module
module -- The file is a module
lnt -- This file is a LNT file
lhdr -- a library header
hdr -- a header file
lmod -- a library module
vcp -- a vcproj file
bpr -- A Borland project file
byp -- an invariant header
twice -- file has special dispensation to be read twice
pch -- a header that should be pre-compiled (a text file)
lph -- a precompiled header (a binary file)
line -- "included" with the **#line** directive
guarded -- has an **#include** guard
noguard -- does not have an **#include** guard

17.4.2 Symbol Flags

defined -- a definition was seen
function_parameter -- a function parameter
library -- comes from a library
referenced -- was referenced somewhere in the program
accessed -- was accessed somewhere in the program
temporary -- is a temporary **rvalue**
mutable -- declared with ’mutable’
register -- defined with ’register’
exported -- was "exported" (e.g., with "**dllexport**")
weak_definial_local -- a weak definial, introduced locally
weak_definial_lib -- a weak definial, introduced in library
weak_definial_proj -- a "project-wide" weak definial
weak_definial_refd -- a referenced "project-wide" weak definial
builtin -- a built-in symbol. (for intrinsic functions, etc.)
class -- declared with ’**class**’ (not ’**struct**’)
func_pure -- causes no side effects
func_impure -- causes side effects
ctor -- a constructor
dtor -- a destructor
virtual -- declared with ’virtual’
private -- declared with ’private’
protected -- declared with ’protected’
op_assign -- **operator=()**
op_delete -- **operator delete()**
op_delete_arr -- **operator delete[]()**
ctor_default -- **A::A()**
ctor_copy -- **A::A(const A&)**
inline -- declared with ’inline’
func_template_spec -- a function template specialization
c -- a symbol with C language linkage
explicit_spec -- an explicit specialization
defined_in_class -- a member function defined in its class
nothrow -- a function that does not throw
constant -- a compile-time constant
template_sp_member -- member of a specialization; e.g. **A<int>::m**
explicit -- declared with **explicit**
throw -- a function that throws

17.4.3 Macro Flags

function_like -- a function-like macro
referenced -- was referenced somewhere in the program
defined -- introduced by ’**#define**’, (not with **-d**)

undefined -- a macro that was **#undef**'d or only defined implicitly (e.g. '**#if A**', where **A** is not defined)
library -- first defined in a library
weak_definial_local -- a weak definial introduced locally
weak_definial_lib -- a weak definial introduced in library
weak_definial_proj -- a "project-wide" weak definial
weak_definial_refd -- a referenced "project-wide" weak definial

17.4.4 Output All Flags

If you want to get all flags without explicitly specifying them in the output format, you can use the shorthand:

```
%[*DELIMITER]
```

where ***DELIMITER*** is the string you want to use to separate flags. Consider the following example. Assume you have within a **.lnt** file:

```
+program_info( output_prefix=,
    format_symbol="%[NAME], flags: {%[*,]}\n"
    )
```

Then assume you have in a C++ source file:

```
int alpha( int ) { return 0; }
```

In the generated **symbol.txt** you will find:

```
alpha, flags: {defined,func_pure}
```

17.5 Output Filtering

The sub-option

```
filter_category( flag [ , flag ...] )
```

may be used to suppress the output of all records bearing the specified flags (See **Section 17.4** above). E.g.:

```
-program_info(
-filter_symbol( private, weak_definial_local, c )
)
```

filters out records of symbols that have **`private`** access, that are weak definials not defined in a project header, or are symbols with C language linkage. The effect of each subsequent **`filter_`*`category`*** sub-option is cumulative. E.g.:

```
-program_info( -filter_symbol( weak_definial_local ) )
-program_info( -filter_symbol( private, c ) )
```

This has the same effect as the previous example.

By default, Lint behaves as if **`-filter_symbol( library )`** had been given. To remove a filter (and thus enable output for symbols with a named flag), use **`+filter_symbol`** as in

```
-program_info( +filter_symbol( library ) )
```

18. COMMON PROBLEMS

18.1 Option has no effect

One common mistake is to place lint options in a comment in a C/C++ program and forget to place a **lint** there or to place a blank before the word **lint**. Examples:

```
/*  -e501   */        Bad!
/* lint  -e501  */    Bad!
/*lint   -e501  */    OK!
```

Another more likely possibility is that the message you are trying to inhibit is delivered at wrap-up time. For example:

```
/*lint -save -e714   This variable is not referenced */
int  xx;
/*lint -restore */
```

This doesn't work because message **714** is issued sometime later (at wrap-up time). In all such cases you should be able to inhibit the message using **-esym**. E.g.:

```
/*lint -esym(714,xx)  xx is not referenced */
int  xx;
```

18.2 Order of option processing

Options are processed in order. For example

```
lint alpha beta -idirectory
```

will process **alpha** and **beta** without benefit of the include directory.

18.3 Too many messages

It should be emphasized that suppressing a message does not alter the behavior of PC-lint/ FlexeLint other than to suppress the message. For example, inhibiting message **718** (function used without a prior declaration) does not inhibit other messages about the function such as "inconsistent return value" or "inconsistent parameters". It is as if you had edited the output file and removed all references to message **718**. So you needn't be too hesitant about inhibiting a message.

To set a warning level, use option **-w**, or **-wlib** (See Section 5.7 Other Options.)

18.4 What is the preprocessor doing?

In order to understand some diagnostics it may be necessary to look at the output of the preprocessor. The option **-p** turns PC-lint/FlexeLint into a preprocessor. For example:

```
lin -p alpha.cpp
```

will produce (by default in **_LINT.TMP**) the result of preprocessing module **alpha.cpp**

This is often sufficient to resolve difficulties. However, in some cases, it is necessary to relate the output of the preprocessor to the header files used in the generation of the output. Which line of which header produced a particular line? To answer this sort of question use the **-v1** option in conjunction with the **-p** option as in

```
lin -v1 -p alpha.cpp
```

This will provide a line-by-line description as to how the headers relate to the output.

18.5 NULL not defined

If you use the Microsoft C compiler and you simply lint a program as:

```
lint  program.c
```

you may get **NULL** not defined. You have to indicate to lint that you are using the Microsoft C/C++ compiler. This is normally done as:

```
lint co-msc.lnt program.c
```

At the very least use:

```
lint -cmsc program.c
```

although this does nothing to describe your compiler's library.

18.6 Error 123 using min or max

Some Microsoft C users have been confused about getting error **123** when "all they do" is have a declaration of the form:

```
int min;  OR  int max;
```

Actually, somewhere in the module is an include of **stdlib.h**, which defines macros **min()** and **max()**. If you do not want PC-lint/FlexeLint to complain about this dual use ("because they're used all over the place"), simply suppress the message with **-e123** or **-esym(123,min,max)**.

18.7 LONG_MIN macro

PC-lint/FlexeLint will occasionally issue a warning (**501** and/or **569**) when using the **LONG_MIN** macro from your compiler's **limits.h** header file. We offer ***no apologies*** for this warning. We have found the following variations in the definition of **LONG_MIN** among several different compiler vendors.

```
#define LONG_MIN -2147483647         /* OK */
#define LONG_MIN 0x80000000L         /* Warning */
#define LONG_MIN (-2147483647-1)     /* OK */
#define LONG_MIN ((long)0x80000000L) /* OK */
#define LONG_MIN -2147483648L        /* Warning */
#define LONG_MIN (-(2147483647L)-1)  /* OK */
#define LONG_MIN -2147483648         /* Warning */
```

For those that we issue a warning, the quantity is typed **unsigned long** and if you used this type as in:

```
if( n > LONG_MIN ) ...
```

you would find that the test, which should just about always succeed would just about never succeed. Perhaps you should alert your compiler vendor.

18.8 Plain Vanilla Functions

N.B. The following pertains only to C code.

By a plain vanilla function (or canonical function) we mean a function declared without a prototype. For example

```
void f();
```

Not too many programmers realize that such a function is incompatible with one that is prototyped with a **char**, **short**, or **float** parameter or has an ellipsis. We warn you (type difference = '**promotion**' or '**ellipsis**') but the warning can cause confusion if you do not realize the difference.

When a call is made to such a function the compiler must decide which, if any, promotions to apply to the arguments. Since the declaration said nothing about arguments, a standard (i.e.,

canonical) set of promotions is applied. According to ANSI, **char** and **short** are promoted to **int**, and **float** is promoted to **double**. Also the argument list is presumed fixed so that registers may be used to pass arguments.

Prototypes can inhibit such promotions; if **f** was declared:

```
void f( char, short, float );
```

All three promotions would be inhibited. For this reason this declaration is incompatible with the earlier declaration and you receive a warning. If **f** were declared:

```
void f( int, ...);
```

we again warn you because the canonical declaration allows the compiler to pass arguments in registers and the ellipsis forces the compiler to pass arguments on the stack.

This is all in the ANSI/ISO C standard.

18.9 Avoiding Lint Comments in Your Code

Occasionally there is a requirement that there be no lint directives in your source code. A programmer can go pretty far in inhibiting unwanted messages by using the **-e**, **-esym**, etc. options placed within a **.lnt** file but occasionally it happens that a specific occurrence of a message needs to be suppressed. For example:

```
int *pi;
unsigned **ppu;
pi = (int *) ppu;
```

raises message **740** (unusual pointer cast). To suppress this particular usage you may define a macro as in:

```
#define INT_STAR(p)  ((int *) (p))
...
pi = INT_STAR(ppu);
```

You will still get the message, which can then be suppressed with

```
-emacro(740,INT_STAR)
```

18.10 Strange Compilers

You may want to lint programs that have been prepared for compilers that accept strange and unusual constructs, and for which we do not provide a custom compiler options file. There are a number of options you can use to get PC-lint/FlexeLint to ignore such constructs. Chief among these are the **-d**, **+rw** and **+ppw** in Section 5.7 Other Options. But also check Section 5.8.3 Customization Facilities for additional options to help cope with the truly extraordinary.

18.11 !0

If you are using

```
#define TRUE !0
```

you will receive the message:

```
506 -- "Constant Value Boolean"
```

when **TRUE** is used in an arithmetic expression. (For C, **TRUE** should be defined to be 1. However, other languages use quantities other than 1 so some programmers feel that **!0** is playing it safe.) To suppress this message for just this context you can use:

```
#define TRUE /*lint -save -e506 */ (!0) \
                        /*lint -restore */
```

or the equivalent:

```
-emacro(506, TRUE)
```

18.12 What Options am I using?

With options embedded within indirect files and within source code it is sometimes difficult to know what options are in effect. To obtain a listing of all your options, use the verbosity option **-voif** (**o** = Option, **i** = Indirect File, **f** = header Files). To have it take effect early enough to show the options within all indirect files you may set the LINT environment variable as in:

```
set LINT= -voif
lint usual arguments
set LINT=
```

You can also get a peek at some of the option settings by looking at the help screen. For example:

```
lint usual arguments ?
```

This will reveal flag settings and scalar sizes. You might want to redirect the help screen using **-oe(*filename*)** placed before the **?**.

18.13 How do I deal with SQL?

If you have SQL commands of the form

```
EXECSQL   ...   ;
```

embedded in your code, you will find that PC-lint/FlexeLint will stumble over this construct yielding inappropriate messages. To get PC-lint/FlexeLint to ignore such statements, you may define **EXECSQL** to be equivalent to the built-in reserved word **_to_semi** (See Section 5.8.3 Customization Facilities). Don't forget to activate the reserved word. The pair of options needed are:

```
-dEXECSQL=_to_semi
+rw(_to_semi)
```

18.14 Torture Testing Your Code

Ok, this is not a common problem but just thought you might like to know how to maximize the number of messages coming out of PC-lint/FlexeLint. There are several things you can do:

+fsc (String constants are const char flag) assumes string constants are **const char***.

-passes(6) make sure you use plenty of passes as a sequence of calls can have a ripple effect.

+fpn (Pointer parameter may be Null flag) warns about the use of pointer parameters without first checking for NULL.

+fnr (Null can be Returned flag) Any pointer returned by any function is assumed to possibly be Null

-strong(AJX) All typedefs must match exactly.

-w4 Set warning to the max. (This will probably be more torture than you can take - you've been warned).

18.15 Cautions with make

Users of 'make' programs may be surprised to see Lint output something like:

```
--- Module:   /usr/local/bin/flint
```

This happens when the make target for Linting contains:

```
lint_proj:
      $(LINT) $(LINT_OPTIONS) $(INCLUDES) $(SOURCES) > $(LINT_LOG)
```

Users of '**make**' might assume the make variable '**LINT**' should hold the path to the Lint executable; it follows in the tradition of other standard make variable names like '**CC**', '**CXX**', etc. The problem is that make variables may also be environment variables, and Lint uses the environment variable '**LINT**' as a prefix to the list of command line arguments (See **Chapter 4.**). The solution is simply to use some variable name other than '**LINT**' (e.g. '**LINT_EXE**').

19. MESSAGES

Most error messages have an associated error number. By looking up the number in the list below you can obtain additional information about the cause of the error. This information is also available as a machine-readable ASCII file **`msg.txt`**.

Messages numbered **1000** and higher pertain generally to C++. This is summarized in the table below.

After a possible 1000 is subtracted off, the remainder lies in the range 0-999. Remainders in the range 1-199 are syntax errors, 200-299 are PC-lint/FlexeLint internal errors and should never occur, 300-399 are fatal errors usually brought about by exceeding some limit, 400-699 are warning messages that indicate that something is likely to be wrong with the program being examined. Remainders in the range 700-899 designate informational messages. These may be errors but they also may represent legitimate programming practices depending upon personal programming style. Remainders in the range 900-999 are called "Elective Notes". They are not automatically output. You may examine the list to see if you wish to be alerted to any of them.

	C	C++	Warning Level
Syntax Errors	**1 - 199**	**1001 - 1199**	**1**
Internal Errors	**200 - 299**	**1200 - 1299**	
Fatal Errors	**300 - 399**		
Warnings	**400 - 699**	**1400 - 1699**	**2**
Informational	**700 - 899**	**1700 - 1899**	**3**
Elective Notes	**900 - 999**	**1900 - 1999**	**4**

Glossary

A few of the terms used in the commentary below are:

`argument` The actual argument of a function as opposed to a dummy (or formal) parameter of a function (see **`parameter`** below).

`arithmetic` any of the integral types (see below) plus **`float`**, **`double`**, and **`long double`**.

`Boolean` In general, the word Boolean refers to quantities that can be either true or false. An expression is said to be Boolean (perhaps it would be better to say 'definitely Boolean') if it is of the form: ***`operand op operand`*** where ***`op`*** is a relational (**`> >= < <=`**), an equality operator (**`== !=`**), logical *And* (**`&&`**) or logical *Or* (**`||`**). A context is said to require a Boolean if it is used in an **`if`** or **`while`** clause or if it is the 2nd expression of a **`for`** clause or if it is an argument to one of the operators: **`&&`** or **`||`**. An expression needn't be

definitely Boolean to be acceptable in a context that requires a Boolean. Any integer or pointer is acceptable.

declaration gives properties about an object or function (as opposed to a definition).

definition that which allocates space for an object or function (as opposed to a declaration) and which may also indicate properties about the object. There should be only one definition for an object but there may be many declarations.

integral a type that has properties similar to integers. These include **`char`**, **`short`**, **`int`**, and **`long`** and the **`unsigned`** variations of any of these.

scalar any of the arithmetic types plus pointers.

lvalue is an expression that can be used on the Left hand side of an assignment operator (**`=`**). Some contexts require lvalues such as autoincrement (**`++`**) and autodecrement (**`--`**).

macro an abbreviation defined by a **`#define`** statement. It may or may not have arguments.

member elements of a **`struct`** and of a **`union`** are called members.

module that which is compiled by a compiler in a single independent compilation. It typically includes all the text of a **`.c`** (or a **`.cpp`** or **`.cxx`**, etc.) file plus any text within any **`#include`** file(s).

parameter A formal parameter of a function as opposed to an actual argument (see **`argument`** above).

Message Parameters

Some of the messages are parameterized with one or more of the following italicized names:

Message parameters marked with a **`*`** may be used as an argument to **`esym()`** for error suppression or enabling when the **`fsn`** flag is ON.

Char Some character

Context* Specifies one of several contexts in which an assignment can be made. Can be one of:

- **`assignment`** -- refers to an explicit assignment operator.
- **`return`** -- refers to the implied assignment of a **`return`** statement. The type of the expression is converted implicitly to the type of the function.

- **initialization** -- refers to the assignment implied by an initialization statement.
- **arg. no.**... -- refers to the implied assignment of an argument in the presence of a prototype. The type of the expression is implicitly converted to the type within a prototype.
- **arg. 'this'** -- refers to the implied argument of a member function call.

FileName A filename. Messages containing this parameter can be suppressed with the **-efile(** ... **)** option.

Integer Some integer

Invocation A function call with argument types. To suppress a message containing an Invocation you need the complete call, not just the function name. For example, the call **f(1)** could result in Error **1024** parameterized by '**f(int)**'. To suppress this message you could use **-esym(1024,f(int))**. You could also use wild cards as in **-esym(1024,f*)**.

Kind* A list of control structures.

Location A line number followed optionally by a filename (if different from the current) and/or a module name if different from the current.

Name A *string*, usually an identifier, that can be suppressed with a **-esym** but not with **-elibsym**.

String* A sequence of characters identified further in the message description.

Symbol The name of a user identifier referring to a C or C++ object such as variable, function, structure, etc. Messages containing this parameter can be suppressed with the **-esym(** ... **)** option. For C++, if ***Symbol*** is the name of a function the full function signature (including parameters) is given. Error suppression with **-esym** does not require (nor want) the parameters.

Type* A type or a top type base is provided. A top type base is one of **pointer**, **function**, **array**, **struct**, **union**, or **enum**.

TypeDiff* specifies the way in which one type differs from another. Because of type qualification, function prototypes, and type compounding, it may not be obvious how two types differ. Also, see the **-etd** option to inhibit errors based on type differences. ***TypeDiff*** can be one or more of:

- **basic** -- The two types differ in some fundamental way such as **double** versus **int**.

- **count** -- Two function types differ in the number of arguments.

- **`ellipsis`** -- Two function types differ in that one is prototyped using an ellipsis and the other is not prototyped. See Section 18.8 Plain Vanilla Functions

- **`incomplete`** -- At least one of the types is only partially specified such as an array without a dimension or a function without a prototype.

- **`nominal`** -- The types are nominally different but are otherwise the same. For example, **`int`** versus **`long`** where these are the same size or **`double`** versus **`long double`** where these are the same size. The two types are either both integral or both float or are functions that return types or have arguments that differ nominally. If **`long`** is the same size as **`int`** then **`unsigned long`** will differ from **`int`** both as **`nominal`** and as **`signed/unsigned`**. If not the same size, then the difference is **`precision`**.

- **`origin`** -- The types are not actually different but have different origins. For example a **`struct`** is defined in two separate modules rather than in one header file. If for some reason you want to do this then use the option **`-etd(origin)`**.

- **`precision`** -- Two arithmetic types differ in their precision such as **`int`** vs. **`long`** where these are different sizes.

- **`promotion`** -- Two function types differ in that one is prototyped with a **`char`**, **`short`** or **`float`** type and the other is not prototyped.

- **`ptrs to`**... -- Pointers point to different types, some *TypeDiff* code follows.

- **`ptrs to incompatible types`** -- Pointers point to types, which in turn differ in **`precision`**, **`count`**, **`size`**, **`ellipsis`** or **`promotion`**.

- **`qualification`** -- Qualifiers such as **`const`**, **`volatile`**, etc. are inconsistent.

- **`signed/unsigned`** -- The types differ in that one is a signed integral type and the other is unsigned of the same size, or they are both functions that return types that differ in this way, or they are both pointers to types that differ in this way.

- **`size`** -- Two arrays differ in array dimension.

- **`strong`** -- Two types differ in that one is strong and the other is not the same strong type.

- **void/nonvoid** -- The two types differ in that one is **void** and the other is not or, more frequently, they are both functions returning types that differ in this respect or pointers to types that differ in this respect.

- **enum/enum --** One type is an **enum**. The other type is a different **enum**.

- **int/enum --** One type is an **enum** and the other is an **int**.

- ***Type* = *Type*** -- The two types in an assignment of some kind differ in some basic way and no more information is available.

- ***Type* vs. *Type*** -- The two types differ in some basic way and no more information is available.

19.1 C Syntax Errors

1 **Unclosed Comment (*Location*)** -- End of file was reached with an open comment still unclosed. The ***Location*** of the open comment is shown.

2 **Unclosed Quote** -- An end of line was reached and a matching quote character (single or double) to an earlier quote character on the same line was not found.

3 **#else without a #if** -- A **#else** was encountered not in the scope of a **#if**, **#ifdef** or **#ifndef**.

4 **Too many #if levels** -- An internal limit was reached on the level of nesting of **#if** statements (including **#ifdef** and **#ifndef**).

5 **Too many #endif's** -- A **#endif** was encountered not in the scope of a **#if** or **#ifdef** or **#ifndef**.

6 **Stack Overflow** -- One of the built-in non-extendable stacks has been overextended. The possibilities are too many nested **#if** statements, **#include** statements (including all recursive **#include** statements), static blocks (bounded by braces) or **#define** replacements.

7 **Unable to open include file: *FileName*** -- ***FileName*** is the name of the include file, which could not be opened. See also flag **fdi** (See Section 5.5 Flag Options), option **-i**... (See Section 5.7 Other Options) and Section 15.2.1 INCLUDE Environment Variable.

8 **Unclosed #if (*Location*)** -- A **#if** (or **#ifdef** or **#ifndef**) was encountered without a corresponding **#endif**. ***Location*** is the location of the **#if**.

9 **Too many #else's in #if (*Location*)** -- A given **#if** contained a **#else**, which in turn was followed by either another **#else** or a **#elif**. The error message gives the line of the **#if** statement that started the conditional that contained the aberration.

10 **Expecting *String*** -- ***String*** is the expected token. The expected token could not be found. This is commonly given when certain reserved words are not recognized.

```
int __interrupt f();
```

will receive an **Expecting ';'** message at the **f** because it thinks you just declared **__interrupt**. The cure is to establish a new reserved word with **+rw(__interrupt)**. Also, make sure you are using the correct compiler options file. See also Section 18.10 Strange Compilers.

11 **Excessive Size** -- The filename specified on a **#include** line had a length that exceeded **FILENAME_MAX** characters.

12 **Need < or "** -- After a **#include** is detected and after macro substitution is performed, a file specification of the form **<*filename*>** or **"*filename*"** is expected.

13 **Bad type** -- A type adjective such as **long**, **unsigned**, etc. cannot be applied to the type, which follows.

14 **Symbol '*Symbol*' previously defined (*Location*)** -- The named object has been defined a second time. The location of the previous definition is provided. If this is a tentative definition (no initializer) then the message can be suppressed with the **+fmd** flag. (Section 5.5 Flag Options).

15 **Symbol '*Symbol*' redeclared (*TypeDiff*) (*Location*)** -- The named symbol has been previously declared or defined in some other module (location given) with a type different from the type given by the declaration at the current location. The parameter ***TypeDiff*** provides further information on how the types differ (see Glossary in Chapter 19. MESSAGES.).

16 **Unrecognized name** -- A **#** directive is not followed by a recognizable word. If this is not an error, use the **+ppw** option. (Section 5.7 Other Options).

17 **Unrecognized name** -- A non-parameter is being declared where only parameters should be.

18 **Symbol '*Symbol*' redeclared (*TypeDiff*) conflicts with *Location*** -- A symbol is being redeclared. The parameter ***TypeDiff*** provides further information on how the types differ (see Glossary Chapter 19. MESSAGES.). ***Location*** is the

location of the previous definition.

19 **Useless Declaration** -- A type appeared by itself without an associated variable, and the type was not a **struct** and not a **union** and not an **enum**. A double semi-colon can cause this as in:

```
int x;;
```

20 **Illegal use of =** -- A function declaration was followed by an **=** sign.

21 **Expected {** -- An initializer for an indefinite size array must begin with a left brace.

22 **Illegal operator** -- A unary operator was found following an operand and the operator is not a post operator.

23 **Expected colon** -- A **?** operator was encountered but this was not followed by a **:** as was expected.

24 **Expected an expression, found '*String*'** -- An operator was found at the start of an expression but it was not a unary operator.

25 **Illegal constant** -- Too many characters were encountered in a character constant (a constant bounded by **'** marks).

26 **Expected an expression, found '*String*'** -- An expression was not found where one was expected. The unexpected token is placed in the message.

27 **Illegal character (0x*hh*)** -- An illegal character was found in the source code. The hex code is provided in the message. A blank is assumed. If you are using strange characters in identifier names you will get this message for which you may use the **-ident** option. (See Section 5.7 Other Options)

28 **Redefinition of symbol '*Symbol*'*Location*** -- The identifier preceding a colon was previously declared at the ***Location*** given as not being a label.

30 **Expected a constant** -- A constant was expected but not obtained. This could be following a **case** keyword, an array dimension, bit field length, enumeration value, **#if** expression, etc.

31 **Redefinition of symbol '*Symbol*' conflicts with *Location*** -- A data object or function previously defined in this module is being redefined.

32 **Field size (member '*Symbol*') should not be zero** -- The length of a field was given as non-positive, (0 or negative).

33 **`Illegal constant`** -- A constant was badly formed as when an octal constant contains one of the digits **`8`** or **`9`**.

34 **`Non-constant initializer`** -- A non-constant initializer was found for a static data item.

35 **`Initializer has side-effects`** -- An initializer with side effects was found for a static data item.

36 **`Redefining the storage class of symbol '`*`Symbol`*`' conflicts with `*`Location`** -- An object's storage class is being changed.

37 **`Value of enumerator '`*`Symbol`*`' inconsistent (conflicts with `*`Location`*`)`** -- An enumerator was inconsistently valued.

38 **`Offset of symbol '`*`Symbol`*`' inconsistent (`*`Location`*`)`** -- A member of a class or struct appears in a different position (offset from the start of the structure) than an earlier declaration. This could be caused by array dimensions changing from one module to another.

39 **`Redefinition of symbol '`*`Symbol`*`' conflicts with `*`Location`** -- A **`struct`** or **`union`** is being redefined.

40 **`Undeclared identifier '`*`Name`*`'`** -- Within an expression, an identifier was encountered that had not previously been declared and was not followed by a left parenthesis. ***`Name`*** is the name of the identifier.

41 **`Redefinition of symbol '`*`Symbol`*`'`** -- A parameter of either a function or a macro is being repeated.

42 **`Expected a statement`** -- A statement was expected but a token was encountered that could not possibly begin a statement.

43 **`Vacuous type for variable '`*`Symbol`*`'`** -- A vacuous type was found such as the void type in a context that expected substance.

44 **`Need a switch`** -- A **`case`** or **`default`** statement occurred outside a switch.

45 **`Bad use of register`** -- A variable is declared as a **`register`** but its type is inconsistent with it being a **`register`** (such as a function).

46 **`Field type should be int`** -- Bit fields in a structure should be typed **`unsigned`** or **`int`**. If your compiler allows other kinds of objects, such as **`char`**, then simply suppress this message.

47 **`Bad type`** -- Unary minus requires an arithmetic operand.

48 **`Bad type`** -- Unary **`*`** or the left hand side of the ptr (**`->`**) operator requires a pointer operand.

49 **`Expected a type`** -- Only types are allowed within prototypes. A prototype is a function declaration with a sequence of types within parentheses. The processor is at a state where it has detected at least one type within parentheses and so is expecting more types or a closing right parenthesis.

50 **`Attempted to take the address of a non-lvalue`** -- Unary **`&`** operator requires an lvalue (a value suitable for placement on the left hand side of an assignment operator).

51 **`Expected integral type`** -- Unary **`~`** expects an integral type (**`signed`** or **`unsigned char`**, **`short`**, **`int`**, or **`long`**).

52 **`Expected an lvalue`** -- autodecrement (**`--`**) and autoincrement (**`++`**) operators require an lvalue (a value suitable for placement on the left hand side of an assignment operator). Remember that casts do not normally produce lvalues. Thus

```
++(char *)p;
```

is illegal according to the ANSI/ISO standard. This construct is allowed by some compilers and is allowed if you use the **`+fpc`** option (Pointer Casts are lvalues). (See Section 5.5 Flag Options)

53 **`Expected a scalar`** -- Autodecrement (**`--`**) and autoincrement (**`++`**) operators may only be applied to scalars (arithmetics and pointers) or to objects for which these operators have been defined.

54 **`Division by 0`** -- The constant **`0`** was used on the right hand side of the division operator (**`/`**) or the remainder operator (**`%`**).

55 **`Bad type`** -- The context requires a scalar, function, array, or struct (unless **`-fsa`**).

56 **`Bad type`** -- Add/subtract operator requires scalar types and pointers may not be added to pointers.

57 **`Bad type`** -- Bit operators (**`&`**, **`|`** and **`^`**) require integral arguments.

58 **`Bad type`** -- Bad arguments were given to a relational operator; these always require two scalars and pointers can't be compared with integers (unless constant 0).

59 **`Bad type`** -- The amount by which an item can be shifted must be integral.

60 **`Bad type`** -- The value to be shifted must be integral.

61 **`Bad type`** -- The context requires a Boolean. Booleans must be some form of arithmetic or pointer.

62 **`Incompatible types (`*`TypeDiff`*`) for operator ':'`** -- The 2nd and 3rd arguments to **`? :`** must be compatible types.

63 **`Expected an lvalue`** -- Assignment expects its first operand to be an **`lvalue`**. Please note that a cast removes the lvaluedness of an expression. But see also flag **`+fpc`** in Section 5.5 Flag Options.

64 **`Type mismatch (`*`Context`*`) (`*`TypeDiff`*`)`** -- There was a mismatch in types across an assignment (or implied assignment, see ***`Context`***). ***`TypeDiff`*** specifies the type difference. See options **`-epn`**, **`-eps`**, **`-epu`**, **`-epp`** (Section 5.2 Error Inhibition Options) to suppress this message when assigning some kinds of pointers.

65 **`Expected a member name`** -- After a dot (**`.`**) or pointer (**`->`**) operator a member name should appear.

66 **`Bad type`** -- A **`void`** type was employed where it is not permitted. If a **`void`** type is placed in a prototype then it must be the only type within a prototype. (See error number **`49`**.)

67 **`Can't cast from `*`Type`*` to `*`Type`** -- Attempt to cast a non-scalar to an integral.

68 **`Can't cast from `*`Type`*` to `*`Type`** -- Attempt to cast a non-arithmetic to a **`float`**.

69 **`Can't cast from `*`Type`*` to `*`Type`** -- Bad conversion involving incompatible structures or a structure and some other object.

70 **`Can't cast from `*`Type`*` to `*`Type`** -- Attempt to cast to a pointer from an unusual type (non-integral).

71 **`Can't cast from `*`Type`*` to `*`Type`** -- Attempt to cast to a type that does not allow conversions.

72 **`Bad option '`*`String`*`'`** -- Was not able to interpret an option. The option is given in ***`String`***.

73 **`Bad left operand`** -- The cursor is positioned at or just beyond either an **`->`** or a **`.`** operator. These operators expect an expression primary on their left. Please enclose any complex expression in this position within parentheses.

74 **Address of Register** -- An attempt was made to apply the address (**&**) operator to a variable whose storage class was given as register.

75 **Too late to change sizes (option '*String*')** -- The size option was given after all or part of a module was processed. Make sure that any option to reset sizes of objects be done at the beginning of the first module processed or on the command line before any module is processed.

76 **can't open file '*String*'** -- ***String*** is the name of the file. The named file could not be opened for output. The file was destined to become a PC-lint/FlexeLint object module.

77 **Address of bit-field cannot be taken** -- The address of a bit-field cannot be taken. The rules of C only allow for taking the address of a whole byte (a whole **char**).

78 **Symbol '*Symbol*' typedef'ed at *Location* used in expression** -- The named symbol was defined in a **typedef** statement and is therefore considered a type. It was subsequently found in a context where an expression was expected.

79 **Bad type for % operator** -- The **%** operator should be used with some form of integer.

80 **This use of ellipsis is not strictly ANSI** -- The ellipsis should be used in a prototype only after a sequence of types not after a sequence of identifiers. Some compilers support this extension. If you want to use this feature suppress this message.

81 **struct/union not permitted in equality comparison** -- Two **struct**'s or **union**'s are being compared with one of **==** or **!=**. This is not permitted by the ANSI/ISO standard. If your compiler supports this, suppress this message.

82 **return <exp>; illegal with void function** -- The ANSI/ISO standard does not allow an expression form of the **return** statement with a **void** function. If you are trying to cast to **void** as in **return (void)f();** and your compiler allows it, suppress this message.

83 **Incompatible pointer types with subtraction** -- Two pointers being subtracted have indirect types which differ. You can get PC-lint/FlexeLint to ignore slight differences in the pointers by employing one or more of the **-ep**... options described in Section 5.2 Error Inhibition Options.

84 **sizeof object is zero or object is undefined** -- A **sizeof** returned a 0 value. This could happen if the object were undefined or incompletely defined. Make sure a complete definition of the object is in scope when you use **sizeof**.

85 **Array '*Symbol*' has dimension 0** -- An array (named ***Symbol***) was declared

without a dimension in a context that required a non-zero dimension.

86 **Structure '*Symbol*' has no data elements** -- A structure was declared (in a C module) that had no data members. Though legal in C++ this is not legal C.

87 **Expression too complicated for #ifdef or #ifndef** -- By the rules of C there should be only a single identifier following a **#ifdef** or a **#ifndef**. You may also supply a validly constructed C (or C++) comment.

88 **Symbol '*Symbol*' is an array of empty elements** -- An array was declared (in a C module) whose elements were each of 0 length. Though legal in C++ this is not permitted C.

89 **Argument or option too long ('*String*')** -- The length of an option (shown in ***String***) exceeds an internal limit. Please try to decompose the option into something smaller. At this writing the limit is 610 characters.

90 **Option '*Name*' is only appropriate within a lint comment** -- The indicated option is not appropriate at the command or the **.lnt** level. For example if **-unreachable** is given on the command line you will get this message.

91 **Line exceeds *Integer* characters (use +linebuf)** -- A line read from one of the input files is longer than anticipated. By default the line buffer size is 600 characters. Each time you use the **+linebuf** option you can double this size. The size can be doubled ad infinitum.

92 **Negative array dimension or bit field length (*Integer*)** -- A negative array dimension or bit field length is not permitted.

93 **New-line is not permitted within string arguments to macros** -- A macro invocation contains a string that is split across more than one line. For example:

```
A( "Hello
    World" );
```

will trigger this message. Some compilers accept this construct and you can suppress this message with **-e93** if this is your current practice. But it is more portable to place the string constant on one line. Thus

```
A( "Hello World" );
```

would be better.

95 **Expected a macro parameter but instead found '*Name*'** -- The **#** operator (or the non-standard extension to the **#** operator spelled **#@**) was found within a macro definition but was not immediately followed by a parameter of the macro as is required

by the standards. ***Name*** identifies the token immediately to the right of the operator.

96 **Unmatched left brace for *String* on *Location*** -- The purpose of this message is to report the location of a left curly brace that is unmatched by a right curly brace. Such an unmatched left curly can be far removed from the point at which the unbalance was detected (often the end of the compilation unit). Providing the location of the left curly can be extremely helpful in determining the source of the imbalance.

98 **Recovery Error (*String*)** -- A recovery error is issued when an inconsistent state was found while attempting to recover from a syntactic error. The ***String*** provided in the message serves as a clue to this inconsistent state. Since the presumptive cause of the error is an earlier error, priority should be placed on resolving the original error. This "Recovery Error" is meant only to provide additional information on the state of the parser.

101 **Expected an identifier** -- While processing a function declarator, a parameter specifier was encountered that was not an identifier, whereas a prior parameter was specified as an identifier. This is mixing old-style function declarations with the new-style and is not permitted. For example

```
void f(n,int m)
```

will elicit this message.

102 **Illegal parameter specification** -- Within a function declarator, a parameter must be specified as either an identifier or as a type followed by a declarator.

103 **Unexpected declaration** -- After a prototype, only a comma, semi-colon, right parenthesis or a left brace may occur. This error could occur if you have omitted a terminating character after a declaration or if you are mixing old-style parameter declarations with new-style prototypes.

104 **Conflicting types** -- Two consecutive conflicting types were found such as **int** followed by **double**. Remove one of the types!

105 **Conflicting modifiers** -- Two consecutive conflicting modifiers were found such as **far** followed by **near**. Remove one of the modifiers!

106 **Illegal constant** -- A string constant was found within a preprocessor expression as in

```
#if ABC == "abc"
```

Such expressions should be integral expressions.

107 **Label '*Symbol*' (*Location*) not defined** -- The ***Symbol*** at the given ***Location*** appeared in a goto but there was no corresponding label.

108 **Invalid context** -- A **continue** or **break** statement was encountered without an appropriate surrounding context such as a **for**, **while**, or **do** loop or, for the **break** statement only, a surrounding **switch** statement.

109 **The combination 'short long' is not standard, 'long' is assumed** -- Some compilers support the non-standard sequence **short long**. This message reports, as an error, that this sequence is being used. If you are required to use the construct then simply suppress this message. As the message indicates, that type will be presumed to be **long**.

110 **Attempt to assign to void** -- An attempt was made to assign a value to an object designated (possibly through a pointer) as **void**.

111 **Assignment to const object** -- An object declared as **const** was assigned a value. This could arise via indirection. For example, if **p** is a pointer to a **const int** then assigning to ***p** will raise this error.

113 **Inconsistent enum declaration** -- The sequence of members within an **enum** (or their values) is inconsistent with that of another **enum** (usually in some other module) having the same name.

114 **Inconsistent structure declaration for tag '*Symbol*'** -- The sequence of members within a structure (or **union**) is inconsistent with another structure (usually in some other module) having the same name.

115 **Struct/union not defined** -- A reference to a structure or a **union** was made that required a definition and there is no definition in scope. For example, a reference to **p->a** where **p** is a pointer to a **struct** that had not yet been defined in the current module.

116 **Inappropriate storage class** -- A storage class other than **register** was given in a section of code that is dedicated to declaring parameters. The section is that part of a function preceding the first left brace.

117 **Inappropriate storage class** -- A storage class was provided outside any function that indicated either **auto** or **register**. Such storage classes are appropriate only within functions.

118 **Too few arguments (*Integer*) for prototype '*Name*'** -- The number of arguments provided for a function was less than the number indicated by a prototype in scope.

119 **Too many arguments (*Integer*) for prototype *'Name'*** -- The number of arguments provided for a function was greater than the number indicated by a prototype in scope.

120 **Initialization without braces of dataless type *'Symbol'*** -- There was an attempt to initialize a nested object (e.g., an array element) without braces. Additionally, that object type possesses no data members.

```
class A { public: void f(); };
class B { public: A a; int k; } ;
A a[4] = { {}, {}, {}, {} };            // OK
B b = { , 34 };                         // Error 120
```

121 **Attempting to initialize an object of undefined type *'Symbol'*** -- The initialization of an object was attempted where that object type has no visible definition. For example:

```
class Undefined u = { 5 };
```

122 **Digit (*Char*) too large for radix** -- The indicated character was found in a constant beginning with zero. For example, **08** is accepted by some compilers to represent **8** but it should be **010** or plain **8**.

123 **Macro '*Symbol*' defined with arguments at *Location* this is just a warning** -- The name of a macro defined with arguments was subsequently used without a following '**(**'. This is legal but may be an oversight. It is not uncommon to suppress this message (with **-e123**), because some compilers allow, for example, the macro **max()** to coexist with a variable **max**. (See Section 18.6 Error 123 using min or max).

124 **Pointer to void not allowed** -- A pointer to **void** was used in a context that does not permit **void**. This includes subtraction, addition and the relationals (**> >= < <=**).

125 **Too many storage class specifiers** -- More than one storage class specifier (**static**, **extern**, **typedef**, **register** or **auto**) was found. Only one is permitted.

126 **Inconsistent structure definition '*Symbol*'** -- The named structure (or **union** or **enum**) was inconsistently defined across modules. The inconsistency was recognized while processing a lint object module. Line number information was not available with this message. Alter the structures so that the member information is consistent.

127 **Illegal constant** -- An empty character constant (**''**) was found.

128 **Pointer to function not allowed** -- A pointer to a function was found in an

arithmetic context such as subtraction, addition, or one of the relationals (**> >= < <=**).

129 **declaration expected, identifier '*Symbol*' ignored** -- In a context in which a declaration was expected an identifier was found. Moreover, the identifier was not followed by '**(**' or a '**[**'

130 **Expected integral type** -- The expression in a **switch** statement must be some variation of an **int** (possibly **long** or **unsigned**) or an **enum**.

131 **syntax error in call of macro '*Symbol*' at location *Location*** -- This message is issued when a macro with arguments (function-like macro) is invoked and an incorrect number of arguments is provided. ***Location*** is the location of the start of the macro call. This can be useful because an errant macro call can extend over many lines.

132 **Expected function definition** -- A function declaration with identifiers between parentheses is the start of an old-style function definition (K&R style). This is normally followed by optional declarations and a left brace to signal the start of the function body. Either replace the identifier(s) with type(s) or complete the function with a function body.

133 **Too many initializers for aggregate** -- In a brace-enclosed initializer, there are more items than there are elements of the aggregate.

134 **Missing initializer** -- An initializer was expected but only a comma was present.

135 **comma assumed in initializer** -- A comma was missing between two initializers. For example:

```
int a[2][2] = { { 1, 2 }  { 3, 4 } };
```

is missing a comma after the first right brace (**}**).

136 **Illegal macro name** -- The ANSI/ISO standard restricts the use of certain names as macros. **defined** is on the restricted list.

137 **constant '*Symbol*' used twice within switch** -- The indicated constant was used twice as a **case** within a **switch** statement. Currently only enumerated types are checked for repeated occurrence.

138 **Can't add parent '*Symbol*' to strong type *String*; creates loop** -- An attempt was made to add a strong type parent to a **typedef** type. The attempt is either explicit (with the **-strong** option) or implicit with the use of a **typedef** to a known strong type. This attempt would have caused a loop in the strong parent relationship. Such loops are simply not tolerated.

139 **`Can't take sizeof function`** -- There is an attempt to take the **`sizeof`** a function.

140 **`Type appears after modifier`** -- Microsoft modifiers such as **`far`**, **`_near`**, **`__huge`**, **`_pascal`**, etc. etc. modify the declarator to its immediate right. It therefore should not appear before the type. For example, you should write **`int pascal f(void);`** rather than **`pascal int f(void);`**. Note that **`const`** and **`volatile`** differ from the Microsoft modifiers. They may appear before or after the type. After reporting the error an attempt is made to process the modifiers as the programmer probably intended. See also the **`+fem`** flag in Section 5.5 Flag Options.

141 **`The following option has too many elements: '`*`String`*`'`** -- The indicated option (given by '***`String`***') is too big. It most likely consists of an itemized list that has too many items. You should decompose the large option into two or more smaller options that in sum are equivalent to the one large option.

143 **`Erroneous option:`** ***`String`*** -- An option contained information that was inconsistent with itself or with an earlier option. The ***`String`*** provided in the message explains more fully what the problem is.

144 **`Non-existent return value for symbol '`*`Symbol`*`', compare with`** ***`Location`*** -- An attempt was made to use a non-existent return value of the named function (identified by ***`Symbol`***). It was previously decided that the function did not return a value or was declared with **`void`**.

145 **`Type expected before operator, void assumed`** -- In a context in which a type is expected no type is found. Rather, an operator '**`*`**' or '**`&`**' was encountered. The keyword **`void`** was assumed to have preceded this operator.

146 **`Assuming a binary constant`** -- A constant of the form **`0b`**... was encountered. This was taken to be a binary constant. For example, **`0b100`** represents the value **`4`**. If your compiler supports binary constants you may suppress this message.

147 **`sizeof takes just one argument`** -- An expression of the form **`sizeof(a,b)`** was detected. A second argument is non standard and has been used by some compilers to denote an option to the **`sizeof`** operator. If your compiler has a use for the second argument then suppress this message.

148 **`member '`*`Symbol`*`' previously declared at`** ***`Location`*** -- The indicated member was previously declared within the same structure or union. Although a redeclaration of a function may appear benign it is just not permitted by the rules of the language. One of the declarations should be removed.

149 **`C++ construct '`*`String`*`' found in C code`** -- An illegal construct was found in C code. It looked as though it might be suitable for C++. The quoted string identifies

the construct further.

150 **Token '*String*' unexpected *String*** -- An unexpected token was encountered. The action taken, if any, is identified by the second message parameter.

151 **Token '*Name*' inconsistent with abstract type** -- In a context in which an abstract type is allowed such as within a cast or after a **sizeof**, and after starting to parse the abstract type, an identifier was found. For example:

```
x = (int y) z;
```

152 **Lob base file '*file name*' missing** -- The indicated file has been specified as the base of **lob** production via the option **-lobbase()**. On output, this message is given if the **lob** base is missing. The situation is correctable by simply producing the missing **lob** output. This will not be a problem given the appropriate dependencies in the make file. On input, the most likely cause of this message is an out-of-date base file. A hash code within the **lob** file being read, did not match a similar code already embedded within the base. The input **lob** file should be considered in error and should be regenerated. See Chapter 8. LINT OBJECT MODULES.

153 **Could not create temporary file** -- This message is produced when generating a **lob** output file based upon some lob base file. When the **lob** file is produced, it is first written to a temporary. The temporary is generated by the C library function **tmpnam()**.

154 **Could not evaluate type '*String*', int assumed** -- ***String*** in the message is the second argument to either a **printf_code** option or a **scanf_code** option. When used, it was to be evaluated as a type. Unfortunately the type could not be identified.

155 **Ignoring {…} sequence within an expression, 0 assumed** -- A braced sequence within an expression is a non-standard extension of some compilers (in particular GCC). Internally, we treat such a braced sequence as the equivalent of a constant 0. This means that we may be able to quietly lint such constructions if you merely suppress the message.

156 **Braced initializer for scalar type '*Name*'** -- An example of an initializer that will draw this complaint is as follows.

```
int s[] = { { 1 } };
```

After the compiler has seen the first curly it is expecting to see a number (or other numeric expression). Compilers that strictly adhere to the ISO C and C++ Standards will flag this as ill-formed code.

Note that it is legal (but somewhat arcane) to employ a left curly at the top-level when

initializing an object of scalar type. For example, the following is well-formed:

```
int i = { 0 };// OK; initialize scalar i with 0.
char *t = { "bar" }; // OK; initialize scalar t with a
                     //pointer to a statically allocated
                     //array.
```

Also note: as the example above implies, this message can apply to pointers to arrays of **char**; it does not apply to arrays.

157 **No data may follow an incomplete array** -- An incomplete array is allowed within a struct of a C99 or C++ program but no data is allowed to appear after this array. For example:

```
struct A { int x; int a[]; int b; };
```

This diagnostic is issued when the '**b**' is seen.

158 **Assignment to variable *'Symbol'* (*Location*) increases capability** -- An assignment has been made to a variable that increases capability. A typical capability increase is to remove **const** protection as in the following example:

```
int *p;
const int *q;
p = q;          // Error 158
```

If a capability increase is seen in situations other than an assignment or if the variable is not available, Warning **605** is issued. Please see the description of that message for further information concerning capability increase. See also Informational messages **1776** and **1778**.

159 **enum following a type is non-standard** -- Normally two different types are not permitted within the same type specification; this will ordinarily result in Error **104**. However, some compilers support 'sized' enumerations wherein a scalar type can precede the **enum** keyword. E.g.

```
  char enum color { red, green, blue };
```

When the second type is an **enum** we do not issue a **104** but emit Error **159** instead. By suppressing this message (with **-e159**) such constructs will be supported.

160 **The sequence '({' is non standard and is taken to introduce a GNU statement expression** -- Lint encountered the sequence '**({**' in a context where an expression (possibly a sub-expression) is expected.

```
int n = ({  // Error 160 here
```

```
            int y = foo ();
            int z;
            if (y > 0)
                z = y;
            else z = - y;
            z; })
    // Now n has the last value of z.
```

The primary intention of this message is to alert the user to the non-standard nature of this construct. The typical response is to suppress the message and go on. But a few caveats are in order.

Programmers who intend to work only with C code with the GNU extensions may safely disable this diagnostic but C++ users should think twice. This is partly for the reasons given in GCC's documentation (see the section entitled "Statements and Declarations in Expressions") and partly because the meaning of '({' will change in G++ when its maintainers implement Initializer Lists (a new core language feature that is expected to appear in the 2010 version of the ISO C++ Standard).

161 **Repeated use of parameter *'Symbol'* in parameter list** -- The name of a function parameter was repeated. For example:

```
void f( int n, int m, int n ) {}
```

will cause this message to be issued. Names of parameters for a given function must all be different.

19.2 Internal Errors

200-299 Some inconsistency or contradiction was discovered in the PC-lint/FlexeLint system. This may or may not be the result of a user error. This inconsistency should be brought to the attention of Gimpel Software.

19.3 Fatal Errors

Errors in this category are normally fatal and suppressing the error is normally impossible. However, those errors marked with an asterisk(*) can be suppressed and processing will be continued. For example **-e306** will allow reprocessing of modules.

301 **Stack overflow** -- There was a stack overflow while processing declarations. Approximately 50 nested declarators were found. For example, if a '/' followed by 50 consecutive '*'s were to introduce a box-like comment and if the '/' were omitted, then this message would be produced.

302 **Exceeded Available Memory** -- Main memory has been exhausted.

303 **String too long (try +macros)** -- A single **#define** definition or macro invocation exceeded an internal limit (of 4096 characters). As the diagnostic indicates the problem can be corrected with an option.

304 **Corrupt object file, code *Integer*, symbol=*String*** -- A PC-lint/ FlexeLint object file is apparently corrupted. Please delete the object module and recreate it using the **-oo** option. See Section 8.3 Producing a LOB The special code identifier number as well as a list of symbol names are optionally suffixed to the message as an aid in diagnosing the problem by technical support.

305 **Unable to open module '*file name*'** -- ***file name*** is the name of the file. The named module could not be opened for reading. Perhaps you misspelled the name.

306* **Previously encountered module '*FileName*'** -- ***FileName*** is the name of the module. The named module was previously encountered. This is probably a user blunder.

307 **Can't open indirect file '*FileName*'** -- ***FileName*** is the name of the indirect file. The named indirect file (ending in **.lnt**) could not be opened for reading.

308 **Can't write to standard out** -- **stdout** was found to equal NULL. This is most unusual.

309* **#error ...** -- The **#error** directive was encountered. The ellipsis reflects the original line. Normally processing is terminated at this point. If you set the **fce** (continue on **#error**) flag, processing will continue.

310 **Declaration too long: '*String*...'** (try **+macros**) -- A single declaration was found to be too long for an internal buffer (about 2000 characters). This occurred when attempting to write out the declaration using the **-o**... option. The first 30 characters of the declaration is given in ***String***. Typically this is caused by a very long **struct** whose substructures, if any, are untagged. First identify the declaration that is causing the difficulty. If a **struct** or **union**, assign a tag to any unnamed substructures or subunion. A **typedef** can also be used to reduce the size of such a declaration.

312 **Lint Object Module has obsolete or foreign version id: *Integer*** -- A lint object module was produced with a prior or different version of PC-lint/FlexeLint. Delete the **.lob** file and recreate it using your new version of PC-lint/FlexeLint. The integer provided in this message can help to identify the version.

313 **Too many files** -- The number of files that PC-lint/FlexeLint can process has exceeded an internal limit. The FlexeLint user may recompile his system to increase

this limit. Look for symbol **FSETLEN** in **custom.h**. Currently, the number of files is limited to 6400.

314* **Previously used .lnt file: *FileName*** -- The indirect file named was previously encountered. If this was not an accident, you may suppress this message.

315 **Exceeded message limit (see -limit)** -- The maximum number of messages was exceeded. Normally there is no limit unless one is imposed by the **-limit(*n*)** option. (See Section 5.7 Other Options)

316 **Error while writing to file "*file name*"** -- The given file could not be opened for output.

317 **File encoding, *String*, not currently supported; unable to continue** -- Lint detected a byte order mark at the beginning of a file which indicated the file is encoded in the given format. As of this writing, the only formats supported to any extent are ASCII and UTF-8 (for which Lint presumes ASCII).

321 **Declaration stack overflow** -- An overflow occurred in the stack used to contain array, pointer, function or reference modifiers when processing a declarator.

322* **Unable to open include file *FileName*** -- ***FileName*** is the name of the include file, which could not be opened. Directory search is controlled by options: **-i** (See Section 5.7 Other Options), **+fdi** (Section 5.5 Flag Options) and the **INCLUDE** environment variable (See Section 15.2.1 INCLUE Environment Variable). This is a suppressible fatal message. If option **-e322** is used, Error message **7** will kick in. A diagnostic will be issued but processing will continue.

323 **Token '*String*' too long** -- In attempting to save a token for later reuse, a fixed size buffer was exceeded (governed by the size **M_TOKEN**).

324 **Too many symbols *Integer*** -- Too many symbols were encountered. An internal limit was reached.

325 **Cannot re-open file '*file name*'** -- In the case of a large number of nested includes, files in the outer fringe need to be closed before new ones are opened. These outer files then need to be re-opened. An error occurred when attempting to re-open such a file. If you are experiencing this error it may be owing to a basic inability to **fseek** to a point within a file that was designated by **ftell**. In that case you should increase the number of files that can be open simultaneously. This can be done with **-maxopen**.

326 ***String* '*String...*' too long, exceeds *Integer* characters** -- A string (first 40 characters provided in the message) exceeds some internal limit (provided in the message). There is no antidote to this condition in the form of an option. FlexeLint

customers may recompile with a redefinition of either **M_STRING** (maximum string) or **M_NAME** (maximum name). To override the definition in **custom.h** we suggest recompiling with an appropriate **-dvar=value** option assuming your compiler supports the option.

327 **Bad pipe, code *Integer*** -- Communication between a potential front end and PC-lint is done via pipes. If this communication goes awry this message is issued. If you receive the message bring the message number and code number to the attention of the vendor.

328 **Bypass header '*Name*' follows a different header sequence than in module '*String*' which includes *File1* where the current module includes *File2*** -- This message is issued when a header is **#include**'d that had previously been designated as bypass and it has been determined that this header follows a different header include sequence than in some other module. The name of the other module is given by the second parameter of this message. In order not to bury the programmer under a ton of header names, we have made an effort to determine the precise point where the two modules went their separate ways. The first include file difference occurred when that other module included the header identified by ***File1***, whereas the current module was attempting to include the header identified by ***File2***. Each ***Filei*** is a pair of parameters of the form '***String***' (***Location***) where the location is the point of the **#include**.

For example:

```
Module x.cpp:
   #include "alpha.h"
   #include "delta.h"
   #include "beta.h"
   #include "gamma.h"

Module y.cpp:
   #include "alpha.h"
   #include "beta.h"
   #include "gamma.h"
```

When the include of "**beta.h**" occurs in module **y.cpp** (and if **beta.h** has been designated as bypass) there will be a Fatal Error **328** that the header sequence of module '**x.cpp**' differs from the current module in that the former module included '**delta.h**' at a point where the current module included '**beta.h**'.

It was necessary to make this message a fatal error since attempting to bypass headers that do not follow a consistent header sequence is an act of folly. It is possible to continue on after the **328** in hopes of picking up more inconsistencies in other modules. This can be done using the **+fce** (Continue-on-Error flag).

19.4 C Warning Messages

401 **symbol '*Symbol*' not previously declared static at *Location*** -- The indicated ***Symbol*** declared **static** was previously declared without the **static** storage class. This is technically a violation of the ANSI/ISO standard. Some compilers will accept this situation without complaint and regard the ***Symbol*** as **static**.

402 **static function '*Symbol*' (*Location*) not defined** -- The named ***Symbol*** was declared as a **static** function in the current module and was referenced but was not defined (in the module).

403 **static symbol '*Symbol*' has unusual type modifier** -- Some type modifiers such as **_export** are inconsistent with the **static** storage class.

404 **struct not completed within file '*FileName*'** -- A **struct** (or **union** or **enum**) definition was started within a header file but was not completed within the same header file.

405 **#if not closed off within file '*FileName*'** -- An **#if** construct was begun within a header file (name given) but was not completed within that header file. Was this intentional?

406 **Comment not closed off within file '*FileName*'** -- A comment was begun within a header file (name given) but was not completed within that header file. Was this intentional?

407 **Inconsistent use of tag '*Symbol*' conflicts with *Location*** -- A tag specified as a **union**, **struct** or **enum** was respecified as being one of the other two in the same module. For example:

```
struct tag *p;
union tag *q;
```

will elicit this message.

408 **Type mismatch with switch expression** -- The expression within a **case** does not agree exactly with the type within the **switch** expression. For example, an enumerated type is matched against an **int**.

409 **Expecting a pointer or array** -- An expression of the form ***i*[...]** was encountered where ***i*** is an integral expression. This could be legitimate depending on the subscript operand. For example, if **i** is an **int** and **a** is an array then **i[a]** is legitimate but unusual. If this is your coding style, suppress this message.

410 **size_t not what was expected from fzl and/or fzu, using '*Type*'** -- This warning is issued if you had previously attempted to set the type of **sizeof** by use of the options **+fzl**, **-fzl**, or **-fzu**, and a later **size_t** declaration contradicts the setting. This usually means you are attempting to lint programs for another system using header files for your own system. If this is the case we suggest you create a directory housing header files for that foreign system, alter **size_t** within that directory, and lint using that directory.

411 **ptrdiff_t not what was expected from fdl option, using '*Type*'** -- This warning is issued if you had previously attempted to set the type of pointer differences by use of the **fdl** option and a later **ptrdiff_t** declaration contradicts the setting. See suggestion in Error Message **410**.

412 **Ambiguous format specifier '%X'** -- The format specifier **%X** when used with one of the **scanf** family, is ambiguous. With Microsoft C it means **%lx** whereas in ANSI/ISO C it has the meaning of **%x**. This ambiguous format specification has no place in any serious C program and should be replaced by one of the above.

413 **Likely use of null pointer '*Symbol*' in [left/right] argument to operator '*String*' *Reference*** -- From information gleaned from earlier statements, it appears certain that a null pointer (a pointer whose value is 0) has been used in a context where null pointers are inappropriate. These include: Unary *****, pointer increment (**++**) or decrement(**--**), addition of pointer to numeric, and subtraction of two pointers. In the case of binary operators, one of the words '**left**' or '**right**' is used to designate which operand is null. ***Symbol*** identifies the pointer variable that may be null. See also messages **613** and **794**, and Section 10.2 Value Tracking.

414 **Possible division by 0** -- The second argument to either the division operator (**/**) or the modulus operator (**%**) may be zero. Information is taken from earlier statements including assignments, initialization and tests. See Section 10.2 Value Tracking

415 **access of out-of-bounds pointer ('*Integer*' beyond end of data) by operator '*String*'** -- An out-of-bounds pointer was accessed. ***String*** designates the operator. The parameter '***Integer***' gives some idea how far out of bounds the pointer may be. It is measured in units given by the size of the pointed to object. The value is relative to the last item of good data and therefore should always be greater than zero. For example:

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

results in an overflow message containing the phrase '**1 beyond end of data**'. See Section 10.2 Value Tracking

416 **creation of out-of-bounds pointer ('*Integer*' beyond end of data) by operator '*String*'** -- An out-of-bounds pointer was created. See message **415**

for a description of the parameters ***Integer*** and ***String***. For example:

```
int a[10];
   ...
f( a + 11 );
```

Here, an illicit pointer value is created and is flagged as such by PC-lint/FlexeLint. Note that the pointer **a+10** is not considered by PC-lint/FlexeLint to be the creation of an out-of-bounds pointer. This is because ANSI/ISO C explicitly allows pointing just beyond an array. Access through **a+10**, however, as in ***(a+10)** or the more familiar **a[10]**, would be considered erroneous but in that case message **415** would be issued. See Section 10.2 Value Tracking

417 **integral constant '*String*' has precision *Integer* which is longer than long long int** -- The longest possible integer is by default 8 bytes (see the **+fll** flag and then the **-sll#** option). An integral constant was found to be even larger than such a quantity. For example: **0xFFFF0000FFFF0000F** requires 68 bits and would by default elicit this message. ***String*** is the token in error, and ***Integer*** is the binary precision

418 **Passing null pointer to function '*Symbol*', *Context Reference*** -- A NULL pointer is being passed to a function identified by ***Symbol***. The argument in question is given by ***Context***. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option **-function** or **-sem**. See Section 11.1 Function Mimicry (-function) and Section 11.2.1 Possible Semantics.

419 **Apparent data overrun for function '*Symbol*', argument *Integer* exceeds argument *Integer*** -- This message is for data transfer functions such as **memcpy**, **strcpy**, **fgets**, etc. when the size indicated by the first cited argument (or arguments) exceeds the size of the buffer area cited by the second. The message may also be issued for user functions via the **-function** option. See Section 11.1 Function Mimicry (-function) and Section 11.2.1 Possible Semantics.

420 **Apparent access beyond array for function '*Symbol*', argument *Integer* exceeds *Integer Reference*** -- This message is issued for several library functions (such as **fwrite**, **memcmp**, etc.) wherein there is an apparent attempt to access more data than exist. For example, if the length of data specified in the **fwrite** call exceeds the size of the data specified. The function is specified by ***Symbol*** and the arguments are identified by argument number. See also Section 11.1 Function Mimicry (-function) and Section 11.2.1 Possible Semantics.

421 **Caution -- function '*Symbol*' is considered dangerous** -- This message is issued (by default) for the built-in function **gets**. This function is considered dangerous because there is no mechanism to ensure that the buffer provided as first argument will not overflow. A well known computer virus (technically a worm) was

created based on this defect. Through the **-function** option, the user may designate other functions as dangerous. See also **-deprecate**

422 **Passing to function '*Symbol*' a negative value (*Integer*), *Context Reference*** -- An integral value that appears to be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (***Symbol***), the questionable value (***Integer***) and the argument number (***Context***). The function may be a standard library function designed to accept only positive values such as **malloc** or **memcpy** (third argument), or may have been identified by the user as such through the **-function** or **-sem** options.

The negative integral value may in fact be **unsigned**. Thus:

```
void *malloc( unsigned );
void f()
    {
    int n = -1;
    int *p;
    p = malloc(n);                  // Warning 422
    p = malloc( (unsigned) n );     // Warning 422
    }
```

will result in the warnings indicated. Note that casting the expression does not inhibit the warning.

There is a slight difference in behavior on 32-bit systems versus 16-bit systems. If **long** is the same size as **int** (as in 32-bit systems) the warning is issued based upon the sign bit. If **long** is larger than an **int** (as is true on typical 16-bit systems) the warning is issued if the value was a converted negative as in the examples above. It is not issued if an **unsigned int** has the high-order bit set. This is because it is not unreasonable to **malloc** more that 32,176 bytes in a 16-bit system.

423 **Creation of memory leak in assignment to variable '*Symbol*'** -- An assignment was made to a pointer variable (designated by ***Symbol***), which appeared to already be holding the address of an allocated object, which had not been freed. The allocation of memory, which is not freed, is considered a memory leak.

424 **Inappropriate deallocation (*Name1*) for '*Name2*' data.** -- This message indicates that a deallocation (**free()**, **delete**, or **delete[]**) as specified by ***Name1*** is inappropriate for the data being freed. [12, Item 5**]**

The kind of data (specified by ***Name2***) is one or more of: **malloc**, **new**, **new[]**, **static**, **auto**, **member**, **modified** or **constant**. These have the meanings as described below:

malloc data is data obtained from a call to **malloc**, **calloc** or **realloc.**

new and **new[]** data is data derived from calls to **new**.

static data is either **static** data within a function or external data.

auto data is non-static data in a function.

member data is a component of a structure (and hence can't be independently freed).

modified data is the result of applying pointer arithmetic to some other pointer. E.g.

```
p = malloc(100);
free( p+1 );    // warning
```

p+1 is considered modified.

constant data is the result of casting a constant to a pointer. E.g.

```
int *p = (int *) Ox80002;
free(p);    // warning
```

425 **'*Message*' in processing semantic '*String*' at token '*String*'** -- This warning is issued when a syntax error is encountered while processing a semantic option (**-sem**). The '***Message***' depends upon the error. The first '***String***' represents the portion of the semantic being processed. The second '***String***' denotes the token being scanned when the error is first noticed.

426 **Call to function '*Symbol*' violates semantic '*String*'** -- This warning message is issued when a user semantic (as defined by **-sem**) is violated. '***String***' is the subportion of the semantic that was violated. For example:

```
//lint -sem( f, 1n > 10 && 2n > 10 )
void f( int, int );
...
    f( 2, 20 );
```

results in the message:

```
Call to function 'f(int, int)' violates semantic '(1n>10)'
```

427 **// comment terminates in ** -- A one-line comment terminates in the back-slash escape sequence. This means that the next line will be absorbed in the comment (by a standards-conforming compiler -- not all compilers do the absorption, so beware). It is much safer to end the line with something other than a back-slash. Simply tacking on a period will do. If you really intend the next line to be a comment, the line should be started with its own double slash (**//**).

428 **negative subscript (*Integer*) in operator '*String*'** -- A negative integer was added to an array or to a pointer to an allocated area (allocated by **malloc**, **operator new**, etc.) This message is not given for pointers whose origin is unknown since a negative subscript is, in general, legal.

The addition could have occurred as part of a subscript operation or as part of a pointer arithmetic operation. The operator is denoted by ***String***. The value of the integer is given by ***Integer***.

429 **Custodial pointer '*Symbol*' (*Location*) has not been freed or returned** -- A pointer of **auto** storage class was allocated storage, which was neither freed nor returned to the caller. This represents a "memory leak". A pointer is considered custodial if it uniquely points to the storage area. It is not considered custodial if it has been copied. Thus:

```
int *p = new int[20];  // p is a custodial pointer
int *q = p;            // p is no longer custodial
p = new int[20];       // p again becomes custodial
q = p + 0;             // p remains custodial
```

Here **p** does not lose its custodial property by merely participating in an arithmetic operation.

A pointer can lose its custodial property by passing the pointer to a function. If the parameter of the function is typed pointer to **const** or if the function is a library function, that assumption is not made. For example

```
p = malloc(10);
strcpy (p, "hello");
```

Then **p** still has custody of storage allocated.

It is possible to indicate via semantic options that a function will take custody of a pointer. See **custodial(*i*)** in Section 11.2.1 Possible Semantics. It is possible to declare that no functions take custody other than those specified in a **-sem** option. See also Flag **ffc** (Functions take custody).

430 **Character '@', taken to specify variable location, is not standard C/C++** -- Many compilers for embedded systems have a declaration syntax that specifies a location in place of an initial value for a variable. For example:

```
int x @0x2000;
```

specifies that variable **x** is actually location **0x2000**. This message is a reminder that this syntax is non-standard (although quite common). If you are using this syntax on purpose, suppress this message.

431 **Missing identifier for template parameter number *Integer*** -- A template object parameter (as opposed to a type parameter) was not provided with an identifier. Was this an oversight?

432 **Suspicious argument to malloc** -- The following pattern was detected:

```
malloc( strlen(e+1) )
```

where ***e*** is some expression. This is suspicious because it closely resembles the commonly used pattern:

```
malloc( strlen(e)+1 )
```

If you really intended to use the first pattern then an equivalent expression that will not raise this error is:

```
malloc( strlen(e)-1 )
```

433 **Allocated area not large enough for pointer** -- An allocation was assigned to a pointer whose reach extends beyond the area that was allocated. This would usually happen only with library allocation routines such as **malloc** and **calloc**. For example:

```
int *p = malloc(1);
```

This message is also provided for user-declared allocation functions. For example, if a user's own allocation function is provided with the following semantic:

```
-sem(ouralloc,@P==malloc(1n))
```

We would report the same message. Please note that it is necessary to designate that the returned area is freshly allocated (ala **malloc**).

This message is always given in conjunction with the more general Informational Message **826**.

434 **White space ignored between back-slash and new-line** -- According to the C and C++ standards, any back-slash followed immediately by a new-line results in the deletion of both characters. For example:

```
#define A  \
        34
```

defines **A** to be **34**. If a blank or tab intervenes between the back-slash and the new-line then according to a strict interpretation of the standard you have defined **A** to be a back-slash followed by blank or tab. But this blank is invisible to the naked eye and hence could lead to confusion. Worse, some compilers silently ignore the white-space and the program becomes non-portable.

You should never deliberately place a blank at the end of a line and any such blanks should be removed. If you really need to define a macro to be back-slash blank you can use a comment as in:

```
#define A \   /* commentary */
```

435 **integral constant '*String*' has precision *Integer*, use +fll to enable long long"** -- An integer constant was found that had a precision that was too large for a **long** but would fit within a **long long**. Yet the **+fll** flag that enables the **long long** type was not set.

Check the sizes that you specified for **long** (**-sl#**) and for **long long** (**-sll#**) and make sure they are correct. Turn on **+fll** if your compiler supports **long long**. Otherwise use smaller constants.

436 **Preprocessor directive in invocation of macro '*Symbol*' at *Location*** -- A function like macro was invoked whose arguments extended for multiple lines, which included preprocessor statements. This is almost certainly an error brought about by a missing right parenthesis.

By the rules of Standard C the preprocessing directive is absorbed into the macro argument but then will not subsequently get executed. For this reason some compilers treat the apparent preprocessor directive as a directive. This is logical but not portable. It is therefore best to avoid this construct.

437 **Passing struct '*Symbol*' to ellipsis** -- A struct is being passed to a function at a parameter position identified by an ellipsis. For example:

```
void g()
    {
    struct A { int a; } x;
    void f( int, ... );
    f( 1, x );
    ...
    }
```

This is sufficiently unusual that it is worth pointing out in the likelihood that this is unintended. The situation becomes more severe in the case of a Non-POD **struct [10]**. In this case the behavior is considered undefined.

438 **Last value assigned to variable '*Symbol*' not used** -- A value had been assigned to a variable that was not subsequently used. The message is issued either at a return statement or at the end of a block when the variable goes out of scope. For example, consider the following function:

```
void f( int n )
```

```
{
int x = 0, y = 1;
if( n > 0 )
    {
    int z;
    z = x + y;
    if( n > z ) { x = 3; return; }
    z = 12;
    }
}
```

Here we can report that **x** was assigned a value that had not been used by the time the return statement had been encountered. We also report that the most recently assigned value to **z** is unused at the point that **z** goes out of scope. See also Informational message **838** and flags **-fiw** and **-fiz**.

440 **for clause irregularity: variable '*Symbol*' tested in 2nd expression does not match '*Symbol*' modified in 3rd** -- A **for** clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd **for** clause expression, does not match the variable that is tested in the 2nd **for** clause expression. For example:

```
for( i = 0; i < 10; j++ )
    ...
```

would draw this complaint since the '**i**' of the 2nd expression does not match the '**j**' of the third expression.

441 **for clause irregularity: loop variable '*Symbol*' not found in 2nd for expression** -- The loop variable is determined by an examination of the 3rd **for** clause expression. A loop variable was found (and its name is given in the message) but it did not appear as one of the accessed symbols of the condition expression (the 2nd **for** expression). For example:

```
for( p = a; *p; j++ )
    ...
```

would draw this complaint since the 2nd expression does not contain the '**j**' of the third expression.

442 **for clause irregularity: testing direction inconsistent with increment direction** -- A **for** clause was encountered that appeared to have a parity problem. For example:

```
for( i = 0; i < 10; i-- )
    ...
```

Here the test for **i** less than 10 seems inconsistent with the 3rd expression of the **for** clause which decreases the value of **i**. This same message would be given if **i** were being increased by the 3rd expression and was being tested for being greater than some value in the 2nd expression.

443 **for clause irregularity: variable '*Symbol*' initialized in 1st expression does not match '*Symbol*' modified in 3rd** -- A **for** clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd **for** clause expression, does not match the variable that is initialized in the 1st expression. For example:

```
for( ii = 0; i < 10; i++ )
    ...
```

would draw this complaint since the '**ii**' of the 1st expression does not match the '**i**' of the third expression.

444 **for clause irregularity: pointer '*Symbol*' incremented in 3rd expression is tested for NULL in 2nd expression** -- The following kind of situation has been detected:

```
for( ... ; p == NULL; p++ )
    ...
```

A loop variable being incremented or decremented would not normally be checked to see if it is NULL. This is more likely a programmer error.

445 **reuse of for loop variable '*Symbol*' at '*Location*' could cause chaos** -- A **for** loop nested within another **for** loop employed the same loop variable. For example:

```
for( i = 0; i < 100; i++ )
    {
    ...
    for( i = 0; i < n; i++ ) { ... }
    }
```

447 **Extraneous whitespace ignored in include directive for file *'FileName'*; opening file *'FileName'*** -- A named file was found to contain either leading or trailing whitespace in the **#include** directive. While legal, the ISO Standards allow compilers to define how files are specified or the header is identified, including the appearance of whitespace characters immediately after the **<** or opening **"** or before the **>** or closing **"**. Since filenames tend not to contain leading or trailing whitespace, Lint ignores the (apparently) extraneous characters and processes the directive as though the characters were never given. The use of a **-efile** option on

either ***FileName*** for this message will cause Lint to process **#include**'s with whitespace intact.

448 **Likely access of pointer pointing *Integer* bytes past nul character by operator '*String*'** -- Accessing past the terminating nul character is often an indication of a programmer error. For example:

```
char buf[20];
strcpy( buf, "a" );
char c = buf[4];   // legal but suspect.
```

Although **buf** has 20 characters, after the **strcpy**, there would be only two that the programmer would normally be interested in.

449 **Pointer variable '*Symbol*' previously deallocated** -- A pointer variable (designated in the message) was freed or deleted in an earlier statement.

451 **Header file '*FileName*' repeatedly included but does not have a standard include guard** -- The file named in the message has already been included in the current module. Moreover it has been determined that this header does not have a standard include guard. A standard include guard has the form

```
#ifndef Name
#define Name
...
#endif
```

with nothing but comments before and after this sequence and nothing but comments between the **#ifndef** and the **#define Name**.

This warning may also be accompanied by a **537** (repeated include header). Message **537** is often suppressed because if you are working with include guards it is not a helpful message. However, the message **451** should be left on in order to check the consistency of the include guards themselves.

See also Elective Note **967**.

452 **typedef Symbol '*Symbol*' redeclared (*TypeDiff*) conflicts with *Location*** -- A **typedef** symbol is being declared to be a different type. This can be legal, especially with multiple modules, but is not good programming practice. It interferes with program legibility.

453 **Function '*Symbol*', previously designated pure, *String 'Name'*** -- A semantic option designated that the named function, ***Symbol***, is pure (lacking non-local side-effects: see the **pure** semantic in Chapter **11. SEMANTICS**). However, an impurity was detected. Such impurities include calling a function through a function

pointer, accessing a volatile variable, modifying a static variable or calling a function whose purity PC-lint/FlexeLint cannot verify. ***String*** describes which of these reasons apply and ***Name*** shows the related variable or function as appropriate.

Despite the inconsistency reported, the function will continue to be regarded as pure.

454 **A thread mutex has been locked but not unlocked** -- A return point in a function has been reached such that a mutex lock that had been previously set has not been unlocked. E.g.,

```
//lint -sem( lock, thread_lock )
void f( int x )
    {
    lock();
    if( x < 0 ) return; // Warning 454
    ...
```

455 **A thread mutex that had not been locked is being unlocked** -- A call to an **unlock()** function was made that was not preceded by a balancing **lock()**. It is assumed that every mutex **lock()** function must be balanced by exactly one **unlock()** function, no more, no less. For example:

```
//lint -sem( lock, thread_lock )
//lint -sem( unlock, thread_unlock )
void f( bool x )
    {
    lock();
    /* something */;
    unlock();
    /* something else */
    unlock();   // Warning 455
    }
```

456 **Two execution paths are being combined with different mutex lock states** -- It is the purpose of this message to make absolutely certain that every lock has a corresponding unlock in the same unbroken sequence of code in the same function.

Execution paths can be combined at the end of an **if** statement, **switch** statement, or the beginning of **while**, **for** and **do** statements, a label (target of **goto**), etc. In all these cases we check to make sure that the mutex lock states are the same. For example:

```
//lint -sem( lock, thread_lock )
void f( bool x )
    {
    if( x ) lock();
    // Warning 456 issued here
```

```
    ...
```

It could be argued that if an **unlock()** call would appear under control of the very same **bool x** in the example above then all would be well. And if this is your coding style you are free to turn this message off. But errors in mutex locking have such horrible programming consequences as to suggest especially strong measures to assure code correctness. We recommend, for example:

```
//lint -sem( lock, thread_lock )
//lint -sem( unlock, thread_unlock )
void f( bool x )
   {
   if( x )
      { lock(); /* something */; unlock(); }
   else
      { /* something */ }
   }
```

If the '**something**' that is being executed is sufficiently complex then it can be made into a function.

457 **Thread *'Symbol1'* has an unprotected write access to variable *'Symbol2'* which is used by thread *'Symbol3'* --** A variable (***Symbol2***) was modified by a thread (***Symbol1***) outside of any recognized mutex lock. It was also accessed by a second thread (***Symbol3***). The latter access may or may not have been protected. If unprotected, a second message will be issued with the roles of ***Symbol1*** and ***Symbol3*** interchanged.

458 **Thread *'Symbol'* has an unprotected read access to variable *'Symbol2'* which is modified by thread *'Symbol3'*** -- A variable identified in the message was accessed (non-modifying) by the first thread outside of any recognized mutex lock. It was also modified by a second thread (***Symbol3***). The modification may or may not have been protected. If unprotected Warning **457** will also be issued.

459 **Function *'Symbol'* whose address was taken has an unprotected access to variable *'Symbol'*--** This message is activated only when it appears that the program has more than one thread. See chapter **12. Multi-Thread Support** to determine what those conditions might be.

If a function's address is taken, we presume that we are unable to determine statically all the locations from which the function may be called and so we presume that any and all threads can call this function and so the function needs to have protected access to every static variable that it might touch.

There are several remedies to such a message. If multiple threads can indeed access this

function, then place a mutex lock in the function. If there already is a mutex lock and we don't recognize it, then set the **thread_protected** semantic for the function. If only one thread really accesses this function or if the access is guaranteed to be benign, then, after making sure this condition is commented in the code, use the same **thread_protected** semantic for the function.

460 **Thread '*Symbol*' has unprotected call to thread unsafe function '*Symbol*' which is also called by thread '*Symbol*'** -- The second symbol in the message represents a function that was designated as being **thread_unsafe** through the **-sem** option. It was being called in an unprotected region of a thread whose root function is the first symbol in the message. Another thread is also accessing this function and this thread is identified by the third parameter of the message.

Calls to thread unsafe functions need to be protected by mutex locks if they are to be employed by more than one thread.

461 **Thread '*Symbol*' has unprotected call to function '*Symbol*' of group '*Name*' while thread '*Symbol*' calls function '*Symbol*' of the same group** -- This message is similar to Warning **460** in that a thread (identified in the message as the first ***Symbol***) is making a call on a function (the second ***Symbol***) which had been deduced (through options) as being thread unsafe. Like message **460** there is another thread that is also doing some calling. In this case the other thread is not calling the same function as the first but one which has been placed within the same group (identified by the third parameter) as the first function. See Chapter **12. Multi-Thread Support** to obtain further information on thread unsafe function groups and options to determine them.

462 **Thread '*Symbol*' calling function '*Symbol*' is inconsistent with the '*String*' semantic --** The first ***Symbol*** in the message identifies a thread. The second ***Symbol*** identifies a function called directly or indirectly by the thread. The ***String*** argument specifies a semantic that had been attributed to the function. It should have one of the following forms:

```
thread_not
thread_not( list )
thread_only( list )
```

If the second form is given, it means that the thread appears on the ***list***. If the 3rd form is given it means that the thread was not on the ***list***.

464 **Buffer argument will be copied into itself** --This is issued when we encounter a function argument expression used in such a way that there will be an attempt to copy its contents onto itself. E.g.

```
sprintf( s, "%s", s );
```

501 **`Expected signed type`** -- The unary minus operator was applied to an unsigned type. The resulting value is a positive unsigned quantity and may not be what was intended.

502 **`Expected unsigned type`** -- Unary **`~`** being a bit operator would more logically be applied to unsigned quantities rather than signed quantities.

503 **`Boolean argument to relational`** -- Normally a relational would not have a Boolean as argument. An example of this is **`a < b < c`**, which is technically legal but does not produce the same result as the mathematical expression, which it resembles.

504 **`Unusual shift operation (`*`String`*`)`** -- Either the quantity being shifted or the amount by which a quantity is to be shifted was derived in an unusual way such as with a bit-wise logical operator, a negation, or with an unparenthesized expression. If the shift value is a compound expression that is not parenthesized, parenthesize it.

505 **`Redundant left argument to comma`** -- The left argument to the comma operator had no side effects in its top-most operator and hence is redundant.

506 **`Constant value Boolean`** -- A Boolean, i.e., a quantity found in a context that requires a Boolean such as an argument to **`&&`** or **`||`** or an **`if()`** or **`while()`** clause or **`!`**, was found to be a constant and hence will evaluate the same way each time.

507 **`Size incompatibility, converting `*`Integer`*` byte pointer to `*`Integer`*` byte integral`** -- A cast was made to an integral quantity from a pointer and according to other information given or implied it would not fit. For example a cast to an **`unsigned int`** was specified and information provided by the options indicate that a pointer is larger than an **`int`**. Two ***`Integers`*** are supplied. The first is the size in bytes of the pointer and the second is the size in bytes of the integer.

508 **`extern used with definition`** -- A function definition was accompanied with an **`extern`** storage class. **`extern`** is normally used with declarations rather than with definitions. At best the **`extern`** is redundant. At worst you may trip up a compiler.

509 **`extern used with definition`** -- A data object was defined with a storage class of **`extern`**. This is technically legal in ANSI/ISO and you may want to suppress this message. However, it can easily trip up a compiler and so the practice is not recommended at this time.

511 **`Size incompatibility`** -- A cast was made from an integral type to a pointer and the size of the quantity was too large to fit into the pointer. For example if a **`long`** is cast to a pointer and if options indicate that a **`long`** is larger than a pointer, this warning would be reported.

512 **`Symbol '`*`Symbol`*`' previously used as static (`*`Location`*`)`** -- The ***`Symbol`***

name given is a function name that was declared as **static** in some other module (the location of that declaration is provided). The use of a name as **static** in one module and external in another module is legal but suspect.

514 **Unusual use of a Boolean operator** -- An argument to an arithmetic operator (**+ - / * %**) or a bit-wise logical operator (**| & ^**) was a Boolean. This can often happen by accident as in:

```
if( flags & 4 == 0 )
```

where the **==**, having higher precedence than **&**, is done first (to the puzzlement of the programmer).

515 **Symbol '*Symbol*' has arg. count conflict (*Integer* vs. *Integer*) with *Location*** -- An inconsistency was found between the number of actual arguments provided in a function call and either the number of formal parameters in its definition or the number of actual arguments in some other function call. See the **+fva** option to selectively suppress this message.

516 **Symbol '*Symbol*' has arg. type conflict (no. *Integer* -- *TypeDiff*) with *Location*** -- An inconsistency was found in the type of an actual argument in a function call with either the type of the corresponding formal parameter in the function definition or the type of an actual argument in another call to the same function or with the type specified for the argument in the function's prototype. The call is not made in the presence of a prototype. See options **-ean**, **-eau**, **-eas** and **-eai** in Section 5.2 Error Inhibition Options for selective suppression of some kinds of type differences. If the conflict involves types **char** or **short** then you may want to consider using the **+fxc** or **+fxs** option. (Section 5.5 Flag Options)

517 **defined not K&R** -- The **defined** function (not a K&R construct) was employed and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use **defined**.

518 **Expected '(' -- sizeof** ***type*** is not strict C. **sizeof(*type*)** or **sizeof *expression*** are both permissible.

519 **Size incompatibility** -- An attempt was made to cast a pointer to a pointer of unequal size. This could occur for example in a P model where pointers to functions require 4 bytes whereas pointers to data require only 2. This error message can be circumvented by first casting the pointer to an integral quantity (**int** or **long**) before casting to a pointer.

520 **Highest *String 'Name'* lacks side-effects** -- The first expression of a **for** clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning **522**.

521 **Highest *String 'Name'* lacks side-effects** -- The third expression of a **for** clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning **522**.

522 **Highest *String 'Name'* lacks side-effects** -- If a statement consists only of an expression, it should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). For example, if operator ***** is the built-in operator, the statement ***p++**; draws this message with ***String*** equal to operator and ***Name*** equal to *****. But note that **p++**; does not. This is because the highest operator in the former case is '*****' which has no side effects whereas **p++** does. It is possible for a function to have no side-effects. Such a function is called pure. See the discussion of the pure semantic in **Section 11.2.1**. For example:

```
void f() { int n = 3; n++; }
void g() { f(); }
```

will trigger this message with ***String*** in the message equal to function and ***Name*** equal to **f**.

The definition of pure and impure functions and function calls which have side effects are given in the discussion of the pure semantic in **Chapter 11 Semantics**.

524 **Loss of precision (*Context*) (*Type* to *Type*)** -- There is a possible loss of a fraction in converting from a float to an integral quantity. Use of a cast will suppress this message.

525 **Negative indentation from *Location*** -- The current line was found to be negatively indented (i.e., not indented as much) from the indicated line. The latter corresponds to a clause introducing a control structure, and statements and other control clauses and braces within its scope are expected to have no less indentation. If tabs within your program are other than 8 blanks you should use the **-t** option (See Section 13.3 Indentation Checking).

526 **Symbol '*Symbol*' (*Location*) not defined** -- The named external was referenced but not defined and did not appear declared in any library header file nor did it appear in a Library Module. This message is suppressed for unit checkout (**-u** option). Please note that a declaration, even one bearing prototype information is not a definition. See the glossary at the beginning of this chapter. If the ***Symbol*** is a library symbol, make sure that it is declared in a header file that you're including. Also make sure that the header file is regarded by PC-lint/FlexeLint as a Library Header file. Alternatively, the symbol may be declared in a Library Module. See Section 6.1 Library Header Files and Section 6.2 Library Modules for a further discussion.

527 **Unreachable code at token *Name*** -- A portion of the program cannot be

reached. The first token encountered in that portion of the program appears as the parameter in the message. Because the parameter is designated as ***Name*** the message may be suppressed for selected tokens.

528 **Symbol '*Symbol*' (*Location*) not referenced** -- The named static variable or static function was not referenced in the module after having been declared.

529 **Symbol '*Symbol*' (*Location*) not subsequently referenced** -- The named variable was declared but not referenced in a function.

530 **Symbol '*Symbol*' (*Location*) not initialized** -- An **auto** variable was used before it was initialized.

531 **Field size too large for '*Symbol*'** -- The size given for a bit field of a structure exceeds the size of an **int**.

532 **Return mode of function '*Symbol*' inconsistent with *Location*** -- A declaration (or a definition) of a function implies a different return mode than a previous statement. (The return mode of a function has to do with whether the function does, or does not, return a value). A return mode is determined from a declaration by seeing if the function returns **void** or, optionally, by observing whether an explicit type is given. See the **fdr** flag for a further explanation of this. See also the **fvr** and **fvo** flags in **Section 5.5 Flag Options**.

533 **function '*Symbol*' should (not) return a value (see *Location*)** -- A return statement within a function (or lack of a return at the end of the function) implies a different return mode than a previous statement at ***Location.*** (The return mode of a function has to do with whether the function does, or does not, return a value.) See also the **fvr**, **fvo** and **fdr** flags in Section 5.5 Flag Options.

534 **Ignoring return value of function '*Symbol*' (compare with *Location*)** -- A function that returns a value is called just for side effects as, for example, in a statement by itself or the left-hand side of a comma operator. Try: **(void) function();** to call a function and ignore its return value. See also the **fvr**, **fvo** and **fdr** flags in **Section 5.5 Flag Options**.

537 **Repeated include file '*FileName*'** -- The file whose inclusion within a module is being requested has already been included in this compilation. The file is processed normally even if the message is given. If it is your standard practice to repeat included files then simply suppress this message.

538 **Excessive size** -- The size of an array equals or exceeds 64K bytes.

539 **Did not expect positive indentation from *Location*** -- The current line was found to be positively indented from a clause that did not control the line in

question. For example:

```
if( n > 0 )
   x = 3;
   y = 4;
```

will result in this warning being issued for **`y = 4;`**. The ***`Location`*** cited will be that of the **`if`** clause. See Section 13.3 Indentation Checking

540 **`Excessive size`** -- A string initializer required more space than what was allocated.

541 **`Excessive size`** -- The size of a character constant specified with **`\x`*`ddd`*** or **`\x`*`hhh`*** equaled or exceeded **`2**b`** where **`b`** is the number of bits in a byte (established by the **`-sb`** option). The default is **`-sb8`**.

542 **`Excessive size for bit field`** -- An attempt was made to assign a value into a bit field that appears to be too small. The value to be assigned is either another bit field larger than the target, or a numeric value that is simply too large. You may cast the value to the generic unsigned type to suppress the error.

You may get this message unexpectedly if the base of the bit field is an **`int`**. For example:

```
struct { int b : 1 } s;
s.b = 1;        /* Warning -- requires 0 or -1 */
```

The solution in this case is to use '**`unsigned`**' rather than '**`int`**' in the declaration of **`b`**.

544 **`endif or else not followed by EOL`** -- The preprocessor directive **`#endif`** should be followed by an end-of-line. Some compilers specifically allow commentary to follow the **`#endif`**. If you are following that convention simply turn this error message off.

545 **`Suspicious use of &`** -- An attempt was made to take the address of an array name. At one time such an expression was officially illegal (K&R C **[1]**), was not consistently implemented, and was, therefore, suspect. However, the expression is legal in ANSI/ ISO C and designates a pointer to an array. For example, given

```
int a[10];
int (*p) [10];
```

Then **`a`** and **`&a`**, as pointers, both represent the same bit pattern, but whereas **`a`** is a pointer to **`int`**, **`&a`** is a pointer to an array of 10 integers. Of the two only **`&a`** may be assigned to **`p`** without complaint. If you are using the **`&`** operator in this way, we recommend that you disable this message.

546 **Suspicious use of &** -- An attempt was made to take the address of a function name. Since names of functions by themselves are promoted to address, the use of the **&** is redundant and could be erroneous.

547 **Redefinition of symbol '*Symbol*' conflicts with *Location*** -- The indicated symbol had previously been defined (via **#define**) to some other value.

548 **else expected** -- A construct of the form **if(*e*);** was found, which was not followed by an **else**. This is almost certainly an unwanted semi-colon as it inhibits the **if** from having any effect.

549 **Suspicious cast** -- A cast was made from a pointer to some enumerated type or from an enumerated type to a pointer. This is probably an error. Check your code and if this is not an error, then cast the item to an intermediate form (such as an **int** or a **long**) before making the final cast.

550 **Symbol '*Symbol*' (*Location*) not accessed** -- A variable (local to some function) was not accessed. This means that the value of a variable was never used. Perhaps the variable was assigned a value but was never used. Note that a variable's value is not considered accessed by autoincrementing or autodecrementing unless the autoincrement/decrement appears within a larger expression, which uses the resulting value. The same applies to a construct of the form: ***var += expression***. If an address of a variable is taken, its value is assumed to be accessed. However, casting that address to a non-pointer causes Lint to forget this sense of "accessed-ness." An array, **struct** or **union** is considered accessed if any portion thereof is accessed.

551 **Symbol '*Symbol*' (*Location*) not accessed** -- A variable (declared **static** at the module level) was not accessed though the variable was referenced. See the explanation under message **550** (above) for a description of "access".

552 **Symbol '*Symbol*' (*Location*) not accessed** -- An external variable was not accessed though the variable was referenced. See the explanation under message **550** above for a description of "access".

553 **Undefined preprocessor variable '*Name*', assumed 0** -- The indicated variable had not previously been defined within a **#define** statement and yet it is being used in a preprocessor condition of the form **#if** or **#elif**. Conventionally all variables in preprocessor expressions should be pre-defined. The value of the variable is assumed to be **0**.

555 **#elif not K&R** -- The **#elif** directive was used and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use **#elif**.

556 **indented #** -- A preprocessor directive appeared indented within a line and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not indent the **#**.

557 **unrecognized format** -- The format string supplied to **printf**, **fprintf**, **sprintf**, **scanf**, **fscanf**, or **sscanf** was not recognized. It is neither a standard format nor is it a user-defined format (see **printf_code** and **scanf_code**, Section 5.7 Other Options).

558 **Too few arguments for format (*Integer* missing)** -- The number of arguments supplied to **printf**, **sprintf**, **fprintf**, **scanf**, **fscanf** or **sscanf** was less than the number expected as a result of analyzing the format string. The number of missing arguments is given by ***Integer***. See also message **719**.

559 **size of argument number *Integer* inconsistent with format** -- The given argument (to **printf**, **sprintf**, or **fprintf**) was inconsistent with that which was anticipated as the result of analyzing the format string. Argument counts begin at 1 and include file, string and format specifications. For example,

```
sprintf( buffer, "%f", 371 )
```

will show an error in argument number 3 because constant **371** is not floating point.

560 **argument no. *Integer* should be a pointer** -- The given argument (to one of the **scanf** or **printf** family of functions) should be a pointer. For the **scanf** family, all arguments corresponding to a format specification should be pointers to areas that are to be modified (receive the results of scanning). For the **printf** family, arguments corresponding to **%s** or **%n** also need to be pointers.

Argument counts begin at 1 and include file, string and format specifications. For example

```
scanf( "%f", 3.5 )
```

will generate the message that argument no. 2 should be a pointer.

561 **(arg. no. *Integer*) indirect object inconsistent with format** -- The given argument (to **scanf**, **sscanf**, or **fscanf**) was a pointer to an object that was inconsistent with that which was anticipated as the result of analyzing the format string. Argument counts begin at 1 and include file, string and format specifications. For example if **n** is declared as **int** then:

```
scanf( "%c", &n )
```

will elicit this message for argument number 2.

562 **Ellipsis (...) assumed** -- Within a function prototype a comma was immediately followed by a right parenthesis. This is taken by some compilers to be equivalent to an ellipsis (three dots) and this is what is assumed by PC-lint/FlexeLint. If

your compiler does not accept the ellipsis but makes this assumption, then you should suppress this message.

563 **`Label 'Symbol' (Location) not referenced`** -- The ***`Symbol`*** at the cited ***`Location`*** appeared as a label but there was no statement that referenced this label.

564 **`variable 'Symbol' depends on order of evaluation`** -- The named variable was both modified and accessed in the same expression in such a way that the result depends on whether the order of evaluation is left-to-right or right-to-left. One such example is: **`n + n++`** since there is no guarantee that the first access to **`n`** occurs before the increment of **`n`**. Other, more typical cases, are given in Section 13.1 Order of Evaluation and Section 13.5 volatile Checking. This message is also triggered by the potential modification of an object by a call to a function whose corresponding parameter is a non-**`const`** reference or a non-**`const`** pointer.

565 **`tag 'Symbol' not previously seen, assumed file-level scope`** -- The named tag appeared in a prototype or in an inner block and was not previously seen in an outer (file-level) scope. The ANSI/ISO standard is dubious as to how this tag could link up with any other tag. For most compilers this is not an error and you can safely suppress the message. On the other hand, to be strictly in accord with standard C you may place a small stub of a declaration earlier in the program. For example:

```
struct name;
```

is sufficient to reserve a place for **`name`** in the symbol table at the appropriate level.

566 **`Inconsistent or redundant format char 'Char'`** -- This message is given for format specifiers within formats for the **`printf/scanf`** family of functions. The indicated character ***`Char`*** found in a format specifier was inconsistent or redundant with an earlier character found in the same format specifier. For example a format containing **`"%ls"`** will yield this error with the character **`'s'`** indicated. This is because the length modifier is designed to be used with integral or float conversions and has no meaning with the string conversion. Such characters are normally ignored by compilers.

567 **`Expected a numeric field before char 'Char'`** -- This message is given for format specifiers within formats for the **`printf/scanf`** family of functions. A numeric field or asterisk was expected at a particular point in the scanning of the format. For example: **`%-d`** requests left justification of a decimal integer within a format field. But since no field width is given, the request is meaningless.

568 **`nonnegative quantity is never less than zero.`** -- Comparisons of the form:

```
u >= 0          0 <= u
u <  0          0 >  u
```

are suspicious if **u** is an unsigned quantity or a quantity judged to be never less then 0. See also message **775**.

569 **Loss of information (*Context*) (*Integer* bits to *Integer* bits)** -- An assignment (or implied assignment, see ***Context***) was made from a constant to an integral variable that is not large enough to hold the constant. Examples include placing a hex constant whose bit requirement is such as to require an **unsigned int** into a variable typed as **int**. The number of bits given does not count the sign bit.

570 **Loss of sign (*Context*) (*Type* to *Type*)** -- An assignment (or implied assignment, see ***Context***) is being made from a negative constant into an unsigned quantity. Casting the constant to **unsigned** will remove the diagnostic but is this what you want? If you are assigning all ones to an **unsigned**, remember that **~0** represents all ones and is more portable than **-1**.

571 **Suspicious Cast** -- Usually this warning is issued for casts of the form:

```
(unsigned) ch
```

where **ch** is declared as **char** and **char** is signed. Although the cast may appear to prevent sign extension of **ch**, it does not. Following the normal promotion rules of C, **ch** is first converted to **int**, which extends the sign and only then is the quantity cast to **unsigned**. To suppress sign extension you may use:

```
(unsigned char) ch
```

Otherwise, if sign extension is what you want and you just want to suppress the warning in this instance you may use:

```
(unsigned) (int) ch
```

Although these examples have been given in terms of casting a **char** they will also be given whenever this cast is made upon a signed quantity whose size is less than the casted type. Examples include signed bit fields, expressions involving **char**, and expressions involving **short** when this type is smaller than **int** or a direct cast of an **int** to an **unsigned long** (if **int** is smaller than **long**). This message is not issued for constants or for expressions involving bit operations.

572 **Excessive shift value (precision *Integer* shifted right by *Integer*)** -- A quantity is being shifted to the right whose precision is equal to or smaller than the shifted value. For example,

```
ch >> 10
```

will elicit this message if **ch** is typed **char** and where **char** is less than 10 bits wide (the usual case). To suppress the message in this case you may cast the shifted quantity to a

type whose length is at least the length of the shift value.

The precision of a constant (including enumeration constants) is determined from the number of bits required in its binary representation. The precision does not change with a cast so that **`(unsigned) 1 >> 3`** still yields the message. But normally the only way an expression such as **`1 >> 3`** can legitimately occur is via a macro. In this case use **`-emacro.`**

573 **`Signed-unsigned mix with divide`** -- one of the operands to **`/`** or **`%`** was signed and the other unsigned; moreover the signed quantity could be negative. For example:

```
u / n
```

where **`u`** is unsigned and **`n`** is signed will elicit this message whereas:

```
u / 4
```

will not, even though **`4`** is nominally an **`int`**. It is not a good idea to mix unsigned quantities with signed quantities in any case (a **`737`** will also be issued) but, with division, a negative value can create havoc. For example, the innocent looking:

```
n = n / u
```

will, if **`n`** is -2 and **`u`** is 2, not assign -1 to **`n`** but will assign some very large value.

To resolve this problem, either cast the integer to **`unsigned`** if you know it can never be less than zero or cast the **`unsigned`** to an integer if you know it can never exceed the maximum integer.

574 **`Signed-unsigned mix with relational`** -- The four relational operators are:

```
>   >=   <   <=
```

One of the operands to a relational operator was signed and the other unsigned; also, the signed quantity could be negative. For example:

```
if( u > n ) ...
```

where **`u`** is unsigned and **`n`** is signed will elicit this message whereas:

```
if( u > 12 ) ...
```

will not (even though **`12`** is officially an **`int`** it is obvious that it is not negative). It is not a good idea to mix unsigned quantities with signed quantities in any case (a **`737`** will also be issued) but, with the four relationals, a negative value can produce obscure results. For example, if the conditional:

```
if( n < 0 ) ...
```

is true then the similar appearing:

```
u = 0;
if( n < u ) ...
```

is false because the promotion to unsigned makes **n** very large.

To resolve this problem, either cast the integer to **unsigned** if you know it can never be less than zero or cast the **unsigned** to an **int** if you know it can never exceed the maximum **int**.

575 **enumeration constant exceeds range for integers** -- For many compilers the value of an enumeration constant is limited to those values that can fit within a **signed** or **unsigned int**.

577 **Mixed memory model (option '*String*')** -- The indicated option requested a change to the memory model after part or all of another module was processed. The memory model option should be specified before any module is processed. The most common cause of this error is specifying the memory model after having specified the standard library. This would be a natural error to make if the standard library file were specified via a **LINT** environment variable.

578 **Declaration of symbol '*Symbol*' hides symbol '*Symbol*' (*Location*)** -- A local symbol has the identical name as a global symbol (or possibly another local symbol). This could be dangerous. Was this deliberate? It is usually best to rename the local symbol.

579 **parameter preceding ellipsis has invalid type** -- When an ellipsis is used, the type preceding the ellipsis should not be a type that would undergo a default promotion such as **char**, **short** or **float**. The reason is that many compilers' variable argument schemes (using **stdarg.h**) will break down.

580 **Redeclaration of function '*Symbol*' (hiding *Location*) causes loss of prototype** -- A declaration of a function within a block hides a declaration in an outer scope in such a way that the inner declaration has no prototype and the outer declaration does. A common misconception is that the resulting declaration is a composite of both declarations but this is only the case when the declarations are in the same scope not within nested scopes. If you don't care about prototypes you may suppress this message. You will still receive other type-difference warnings.

581 **Option '*String*' is obsolete and should no longer be used** -- This message is issued whenever we encounter an option that appears to do more harm than good. '*String*' is the option in question.

582 **esym (or emacro) name '*String*' should not contain '(' unless surrounded by double quotes** -- The name provided to **esym** should not contain a **(**. For example, to suppress message **534** when calling **f(int)** use the option **-esym(534,f)** even if **f** is overloaded.

583 **Comparing type '*Type*' with EOF** -- This message is issued when some form of character is compared against the **EOF** macro. **EOF** is normally defined to be **-1**. For example:

```
while( (ch = getchar()) != EOF ) ...
```

If **ch** is defined to be an **int** all is well. If however it is defined to be some form of **char,** then trouble might ensue. If **ch** is an **unsigned char** then it can never equal **EOF**. If **ch** is a **signed char** then you could get a premature termination because some data character happened to be all ones.

Note that **getchar** returns an **int**. The reason it returns an **int** and not a **char** is because it must be capable of returning 257 different values (256 different characters plus **EOF**, assuming an 8-bit character). Once this value is assigned to a char only 256 values are then possible -- a clear loss of information.

584 **Trigraph sequence (??*Character*) detected** --This message is issued whenever a trigraph sequence is detected and the trigraph processing has been turned off (with a **-ftg**). If this is within a string (or character) constant then the trigraph nature of the sequence is ignored. That is, three characters are produced rather than just one. This is useful if your compiler does not process trigraph sequences and you want linting to mirror compilation. Outside of a string we issue the Warning but we do translate the sequence since it cannot make syntactic sense in its raw state.

585 **The sequence (??*character*) is not a valid Trigraph sequence** -- This warning is issued whenever a pair of '**?**' characters is seen within a string (or character) constant but that pair is not followed by a character which would make the triple a valid Trigraph sequence. Did the programmer intend this to be a Trigraph sequence and merely err? Even if no Trigraph were intended it can easily be mistaken by the reader of the code to be a Trigraph. Moreover, what assurances do we have that in the future the invalid Trigraph might not become a valid Trigraph and change the meaning of the string? To protect yourself from such an event you may place a backslash between the '**?**' characters. Alternatively you may use concatenation of string constants. For example:

```
pattern = "(???) ???-????";         // warning 585
pattern = "(?\?\?) ?\?\?-?\?\?\?";  // no warning
#define Q "?"
pattern = "(" Q Q Q ") " Q Q Q "-" Q Q Q Q  // no warning
```

586 ***String*** **'*Name*' is deprecated.** ***String*** -- The ***Name*** has been deprecated by some use of the deprecate option. See **-deprecate** in **Section 5.7**. The first ***String*** is one of the allowed categories of deprecation. The trailing ***String*** is part of the deprecate option and should explain why the facility has been deprecated.

587 **Predicate '*String*' can be pre-determined and always evaluates to *String*** -- The predicate, identified by the first ***String***, (one of greater than, greater than or equal, less than, less than or equal, equal, or not equal), cannot possibly be other than what is indicated by the second ***String*** parameter. For example:

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

would be greeted with the message that '**==**' always evaluates to 'False'.

588 **Predicate '*String*' will always evaluate to *String* unless an overflow occurs** -- The predicate, identified by the first ***String***, cannot possibly be other than what is indicated by the second ***String*** parameter unless an overflow occurred. For example:

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

would be greeted with the message that '**!=**' always evaluates to '**True**'. See also Message **587**.

589 **Predicate '*String*' will always evaluate to *String* assuming standard division semantics** -- The predicate, identified by the first ***String*** parameter, cannot possibly be other than what is indicated by the second ***String*** parameter assuming standard signed integer division semantics. For example:

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

would be greeted with the message that '**==**' always evaluates to '**False**'.

By standard integer division semantics we mean truncation toward zero so that, for example, **-1/4** has quotient 0 and remainder **-1** and not quotient **-1**, remainder **3**. Although the current C standard **[4]** has endorsed this, the current C++ standard **[10]** regards integer division involving negative numbers to be 'implementation defined'.

See also Message **587**.

590 **Predicate '*String*' will always evaluate to *String* assuming standard shift semantics** -- The predicate, identified by the first ***String*** parameter, cannot possibly be other than what is indicated by the second ***String***

parameter assuming standard signed integer shift semantics. For example:

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

would be greeted with the message that the less-than always evaluates to '**False**'. It is true that if you shift 5 left by 29 bits (or 31 bits) you will have in many cases (probably most cases) a negative number but this is not guaranteed. According to the C Standard **[4]**, shifting a positive signed integer (5 in this case) left by a number of places (**n** in this case) is only valid if the type can accommodate **5*(2**n)**. This is equivalent to not shifting a one into or through the sign bit.

As another example:

```
int n; ...
if( (n >> 5) >= 0 ) ...
```

would always be regarded as true. This is because shifting a negative number to the right yields results that are implementation defined.

See also Message **587**.

591 **Variable '*Symbol*' depends on the order of evaluation; it is used/modified through function '*Symbol*' via calls: *String*** -- The indicated variable (given by the first ***Symbol***) was involved in an expression that contained a call of a function (given by the second ***Symbol***) that would use or modify the variable. Further, the order of evaluation of the two is not determinable. For example:

```
extern int n;
void f() { n++; }
int g() { f(); return 1; }
int h() { return n + g(); }    // Warning 591
```

The above code, on the second pass, will elicit the following warning:

```
Warning 591: Variable 'n' depends on the order of evaluation;
it is modified through function 'g(void)' via calls: g() => f()
```

If the function **g()** is called and then **n** is added, you will obtain a different result than if **n** were first evaluated and then the call made.

The programmer should generally rewrite these expressions so that the compiler is constrained to use the intended order. For example if the programmer wanted to use the **n** prior to the call on **g()** it can alter **h()**to the following:

```
int h()
    { int k = n; return k + g(); }
```

This analysis requires two passes; the first pass builds the necessary call trees.

592 **Non-literal format specifier used without arguments --** A **printf**/**scanf** style function received a non-literal format specifier without trailing arguments. For example:

```
char msg[100];
...
printf( msg );
```

This can easily be rewritten to the relatively safe:

```
char msg[100];
...
printf( "%s", msg );
```

The danger lies in the fact that msg can contain hidden format codes. If **msg** is read from user input, then in the first example, a naive user could cause a glitch or a crash and a malicious user might exploit this to undermine system security. Since the unsafe form can easily be transformed into the safe form the latter should always be used.

593 **Custodial pointer *'Symbol'* (*Location*) possibly not freed or returned** -- This is the 'possible' version of message **429**. A pointer of **auto** storage class was allocated storage and not all paths leading to a **return** statement or to the end of the function contained either a **free** or a **return** of the pointer. Hence there is a potential memory leak. For example:

```
void f( int n )
    {
    int *p = new int;
    if( n ) delete p;
    }                   // message 593
```

In this example an allocation is made and, if **n** is 0, no **delete** will have been made.

Please see message **429** for an explanation of "custodial" and ways of regulating when pointer variables retain custody of allocations.

601 **Expected a type for symbol *Symbol*, int assumed** -- A declaration did not have an explicit type. **int** was assumed. Was this a mistake? This could easily happen if an intended comma was replaced by a semicolon. For example, if instead of typing:

```
 double         radius,
                diameter;
```

the programmer had typed:

```
double          radius;
                diameter;
```

this message would be raised.

602 **Comment within comment** -- The sequence **/*** was found within a comment. Was this deliberate? Or was a comment end inadvertently omitted? If you want PC-lint/ FlexeLint to recognize nested comments you should set the Nested Comment flag using the **+fnc** option. Then this warning will not be issued. If it is your practice to use the sequence:

```
/*
/*            */
```

then use **-e602**.

603 **Symbol '*Symbol*' (*Location*) not initialized** -- The address of the named symbol is being passed to a function where the corresponding parameter is declared as pointer to **const**. This implies that the function will not modify the object. If this is the case then the original object should have been initialized sometime earlier.

604 **Returning address of auto variable '*Symbol*'** -- The address of the named symbol is being passed back by a function. Since the object is an **auto** and since the duration of an **auto** is not guaranteed past the **return**, this is most likely an error. You may want to copy the value into a global variable and pass back the address of the global or you might consider having the caller pass an address of one of its own variables to the callee.

605 **Increase in pointer capability (*Context*)** -- This warning is typically caused by assigning a (pointer to **const**) to an ordinary pointer. For example:

```
int *p;
const int *q;
p = q;      /* 605 */
```

The message will be inhibited if a cast is used as in:

```
p = (int *) q;
```

An increase in capability is indicated because the **const** pointed to by **q** can now be modified through **p**. This message can be given for the **volatile** qualifier as well as the **const** qualifier and may be given for arbitrary pointer depths (pointers to pointers, pointers to arrays, etc.).

If the number of pointer levels exceeds one, things get murky in a hurry. For example:

```
const char ** ppc;
char ** pp;
pp = ppc;        /* 605 - clearly not safe */
ppc = pp;        /* 605 - looks safe but it's not */
```

It was not realized by the C community until very recently that assigning **pp** to **ppc** was dangerous. The problem is that after the above assignment, a pointer to a **const char** can be assigned indirectly through **ppc** and accessed through **pp**, which can then modify the **const char**.

The message speaks of an "increase in capability" in assigning to **ppc**, which seems counter intuitive because the indirect pointer has less capability. However, assigning the pointer does not destroy the old one and the combination of the two pointers represents a net increase in capability.

The message may also be given for function pointer assignments when the prototype of one function contains a pointer of higher capability than a corresponding pointer in another prototype. There is a curious inversion here whereby a prototype of lower capability translates into a function of greater trust and hence greater capability (a Trojan Horse). For example, let

```
void warrior( char * );
```

be a function that destroys its argument. Consider the function:

```
void Troy( void (*horse)(const char *) );
```

Troy() will call **horse()** with an argument that it considers precious believing the **horse()** will do no harm. Before compilers knew better and believing that adding in a **const** to the destination never hurt anything, earlier compilers allowed the Greeks to pass **warrior()** to **Troy** and the rest, as they say, is history.

606 **Non-ANSI escape sequence: '*String*'** -- An escape sequence occurred, within a character or string literal, that was not on the approved list, which is:

\' \" \? \\ \a \b \f \n \r \t \v

octal-digits* \x*hex-digits

607 **Parameter '*Symbol*' of macro found within string** -- The indicated name appeared within a string or character literal within a macro and happens to be the same as the name of a formal parameter of the macro as in:

```
#define mac(n) printf( "n = %d,", n );
```

Is this a coincidence? The ANSI/ISO standard indicates that the name will not be replaced but since many C compilers do replace such names, the construction is suspect. Examine the macro definition and if you do not want substitution, change the name of the parameter. If you do want substitution, set the **+fps** flag (Parameter within String) and suppress the message with **-e607**.

608 **Assigning to an array parameter** -- An assignment is being made to a parameter that is typed array. For the purpose of the assignment, the parameter is regarded as a pointer. Normally such parameters are typed as pointers rather than arrays. However if this is your coding style you should suppress this message.

609 **Suspicious pointer conversion** -- An assignment is being made between two pointers, which differ in size (one is **far** and the other is **near**) but are otherwise compatible.

610 **Suspicious pointer combination** -- Pointers of different size (one is **far** and the other is **near**) are being compared, subtracted, or paired (in a conditional expression). This is suspicious because normally pointers entering into such operations are the same size.

611 **Suspicious cast** -- Either a pointer to a function is being cast to a pointer to an object or vice versa. This is regarded as questionable by the language standards. If this is not a user error, suppress this warning.

612 **Expected a declarator** -- A declaration contained just a storage class and a type. This is almost certainly an error since the only time a type without a declarator makes sense is in the case of a **struct**, **union** or **enum** but in that case you wouldn't use a storage class.

613 **Possible use of null pointer '*Symbol*' in [left/right] argument to operator '*String*' *Reference*** -- From information gleaned from earlier statements, it is possible that a null pointer (a pointer whose value is 0) can be used in a context where null pointers are inappropriate. Such contexts include: Unary *****, pointer increment (**++**) or decrement(**--**), addition of pointer to numeric, and subtraction of two pointers. In the case of binary operators, one of the words '**left**' or '**right**' is used to designate which operand is null. ***Symbol*** identifies the pointer variable that may be NULL. See also messages **413** and **794**.

614 **auto aggregate initializer not constant** -- An initializer for an auto aggregate normally consists of a collection of constant-valued expressions. Some compilers may, however, allow variables in this context in which case you may suppress this message.

615 **auto aggregate initializer has side effects** -- This warning is similar to **614**. Auto aggregates (arrays, structures and possibly union) are normally initialized by

a collection of constant-valued expressions without side-effects. A compiler could support side-effects in which case you might want to suppress this message.

616 **`control flows into case/default`** -- It is possible for flow of control to fall into a **`case`** statement or a **`default`** statement from above. Was this deliberate or did the programmer forget to insert a **`break`** statement? If this was deliberate then place a comment immediately before the statement that was flagged as in:

```
case 'a':  a = 0;
   /* fall through */
case 'b':  a++;
```

Note that the message will not be given for a **`case`** that merely follows another **`case`** without an intervening statement. Also, there must actually be a possibility for flow to occur from above. See also message **`825`** and option **`-fallthrough`**.

617 ***`String`*** **`is both a module and an include file`** -- The named file is being used as both an include file and as a module. Was this a mistake? Unlike Error **`306`** (repeated module) this is just a warning and processing of the file is attempted.

618 **`Storage class specified after a type`** -- A storage class specifier (**`static`**, **`extern`**, **`typedef`**, **`register`** or **`auto`**) was found after a type was specified. This is legal but deprecated. Either place the storage class specifier before the type or suppress this message.

619 **`Loss of precision (`*`Context`*`) (Pointer to Pointer)`** -- A **`far`** pointer is being assigned to a **`near`** pointer either in an assignment statement or an implied assignment such as an initializer, a return statement, or passing an argument in the presence of a prototype (***`Context`*** indicates which). Such assignments are a frequent source of error when the actual segment is not equal to the default data segment. If you are sure that the segment of the **`far`** pointer equals the default data segment you should use a cast to suppress this message.

620 **`Suspicious constant (L or one?)`** -- A constant ended in a lower-case letter **`'l'`**. Was this intended to be a one? The two characters look very similar. To avoid misinterpretations, use the upper-case letter '**`L`**'.

621 **`Identifier clash (Symbol '`*`Name`*`' with Symbol '`*`Name`*`' at `*`String`*`)`** -- The two symbols appeared in the same name space but are identical to within the first ***`count`*** characters set by option **`-idlen(`*`count`*`,`*`option`*`)`**. See **`-idlen`** in Section 5.7 Other Options.

622 **`Size of argument no. `*`Integer`*` inconsistent with format`** -- The argument to **`scanf`**, **`fscanf`** or **`sscanf`**, where position is given by ***`Integer`***, was a pointer whose size did not match the format. For example,

```
int far *p;
scanf( "%d", p );
```

will draw this warning (in the default memory model).

623 **redefining the storage class of symbol '*Symbol*' conflicts with *Location*** -- An inter-module symbol was a **typedef** symbol in one module and an ordinary symbol in another module. This is legal but potentially confusing. Is this what the programmer intended?

624 **typedef '*Symbol*' redeclared (*TypeDiff*) (*Location*)** -- A symbol was declared in a **typedef** differently in two different modules. This is technically legal but is not a wise programming practice.

625 **auto symbol '*Symbol*' has unusual type modifier** -- Some type modifiers such as **far**, **near**, **fortran** are inappropriate for **auto** variables.

626 **argument no. *Integer* inconsistent with format** -- The argument to a **printf** (or **fprintf** or **sprintf**) was inconsistent with the format. Although the size of the quantity was appropriate the type was not. You might consider casting the quantity to the correct type. You could also suppress this message, as more flagrant violations are picked up with warning **559**.

627 **(arg. no. *Integer*) indirect object inconsistent with format** -- The type of an argument to **scanf** (or **fscanf** or **sscanf**) was inappropriate to the format. However, the argument was a pointer and it pointed to a quantity of the expected size.

628 **no argument information provided for function '*Symbol*' (*Location*)** -- The named function was called but there was no argument information supplied. Argument information can come from a prototype or from a function definition. This usually happens when an old-style function declaration indicates that the function is in a library but no prototype is given for the function nor is any argument information provided in a standard library file. This message is suppressed if you are producing a lint object module because presumably the object module will be compared with a library file at some later time.

629 **static class for function '*Symbol*' is non standard** -- A **static** class was found for a function declaration within a function. The **static** class is only permitted for functions in declarations that have file scope (i.e., outside any function). Either move the declaration outside the function or change **static** to **extern**; if the second choice is made, make sure that a **static** declaration at file scope also exists before the **extern** declaration. Though technically the construct is not portable, many compilers do tolerate it. If you suppress the message, PC-lint/FlexeLint will treat it as a proper function declaration.

630 **ambiguous reference to symbol '*Name*'** -- If the **+fab** flag is set, then if two

structures containing the same member name (not necessarily different kinds of structures) are embedded in the same structure and a reference to this member name omits one of the intervening (disambiguating) names, this warning is emitted.

631 **tag '*Symbol*' defined differently at *Location*** -- The **struct**, **union** or **enum** tag ***Symbol*** was defined differently in different scopes. This is not necessarily an error since C permits the redefinition, but it can be a source of subtle error. It is not generally a programming practice to be recommended.

632 **Assignment to strong type '*Name*' in context: *Context*** -- An assignment (or implied assignment, ***Context*** indicates which), violates a Strong type check as requested by a **-strong(A...)** option. See Chapter 9. STRONG TYPES.

633 **Assignment from a strong type '*Name*' in context: *Context*** -- An assignment (or implied assignment, ***Context*** indicates which), violates a Strong type check as requested by a **-strong(X...)** option. See Chapter 9. STRONG TYPES.

634 **Strong type mismatch (type '*Symbol*') in equality or conditional** -- An equality operation (**==** or **!=**) or a conditional operation (**? :**) violates a Strong type check as requested by a **-strong(J...)** option. This message would have been suppressed using flags "**Je**". See Chapter 9. STRONG TYPES.

635 **resetting strong parent of type '*Symbol*', old parent == type '*Symbol*'** -- The strong parent of the given ***Symbol*** is being reset. This is being done with a **-parent** option or by a **typedef**. Note that this may not necessarily be an error; you are being alerted to the fact that the old link is being erased. See Chapter 9. STRONG TYPES.

636 **ptr to strong type '*Name*' versus another type** -- Pointers are being compared and there is a strong type clash below the first level. For example,

```
/*lint -strong(J,INT) */
typedef int INT;
INT *p;  int *q;

if( p == q )    /* Warning 636 */
```

will elicit this warning. This message would have been suppressed using flags "**Je**" or "**Jr**" or both.

637 **Expected index type '*Symbol*' for strong type '*Symbol*'** -- This is the message you receive when an inconsistency with the **-index** option is recognized. A subscript is not the stipulated type (the first type mentioned in the message) nor equivalent to it within the hierarchy of types. See Chapter 9. STRONG TYPES and also **+fhx**.

638 **Strong type mismatch for type '*Name*' in relational** -- A relational operation (**>= <= > <**) violates a Strong type check as requested by a **-strong(J...)** option. This message would have been suppressed using flags "**Jr**". See Chapter 9. STRONG TYPES.

639 **Strong type mismatch for type '*Name*' in binary operation** -- A binary operation other than an equality or a relational operation violates a Strong type check as requested by a **-strong(J...)** option. This message would have been suppressed using flags "**Jo**". See Chapter 9. STRONG TYPES.

640 **Expected strong type '*Name*' in Boolean context** -- A Boolean context expected a type specified by a **-strong(B...)**option. See Chapter 9. STRONG TYPES.

641 **Converting enum to int** -- An enumeration type was used in a context that required a computation such as an argument to an arithmetic operator or was compared with an integral argument. This warning will be suppressed if you use the integer model of enumeration (**+fie**) but you will lose some valuable type-checking in doing so. An intermediate policy is to simply turn off this warning. Assignment of **int** to **enum** will still be caught.

This warning is not issued for a tagless **enum** without variables. For example

```
enum {false,true};
```

This cannot be used as a separate type. PC-lint/FlexeLint recognizes this and treats **false** and **true** as arithmetic constants.

642 **Format char '*Char*' not supported by wsprintf** -- This means that you are using an option of the form: **-printf(w...)** and you are using a format character not supported by the Microsoft Windows function **wsprintf**. If you are not really using **wsprintf** but are using the **w** flag to get **far** pointers you should turn this message off.

643 **Loss of precision in pointer cast** -- A **far** pointer was cast to a **near** pointer. Such casts have had disastrous consequences for Windows programmers. If you really need to make such a cast, you can do it in stages. If you cast to a **long** first (i.e., some integral type that can hold the pointer) and then into a shorter value, we don't complain.

644 **Variable '*Symbol*' (*Location*) may not have been initialized** -- An **auto** variable was not necessarily assigned a value before use. See Section 10.1 Initialization Tracking

645 **Symbol '*Symbol*' (*Location*) may not have been initialized** -- An **auto** variable was conditionally assigned a value before being passed to a function

expecting a pointer to a **const** object. See Warning **603** for an explanation of the dangers of such a construct. See Section 10.1 Initialization Tracking

646 **case/default within *Kind* loop; may have been misplaced** -- A **case** or **default** statement was found within a **for**, **do**, or **while** loop. Was this intentional? At the very least, this reflects poor programming style.

647 **Suspicious truncation** -- This message is issued when it appears that there may have been an unintended loss of information during an operation involving **int** or **unsigned int** the result of which is later converted to **long**. It is issued only for systems in which **int** is smaller than **long**. For example:

```
(long) (n << 8)
```

might elicit this message if **n** is **unsigned int**, whereas

```
(long) n << 8
```

would not. In the first case, the shift is done at **int** precision and the high order 8 bits are lost even though there is a subsequent conversion to a type that might hold all the bits. In the second case, the shifted bits are retained.

The operations that are scrutinized and reported upon by this message are: shift left, multiplication, and bit-wise complementation. Addition and subtraction are covered by Informational message **776**.

The conversion to **long** may be done explicitly with a cast as shown or implicitly via assignment, return, argument passing or initialization.

The message can be suppressed by casting. You may cast one of the operands so that the operation is done in full precision as is given by the second example above. Alternatively, if you decide there is really no problem here (for now or in the future), you may cast the result of the operation to some form of **int**. For example, you might write:

```
(long) (unsigned) (n << 8)
```

In this way PC-lint/FlexeLint will know you are aware of and approve of the truncation.

648 **Overflow in computing constant for operation: *'String'*** -- Arithmetic overflow was detected while computing a constant expression. For example, if **int** is 16 bits then **200 * 200** will result in an overflow. ***String*** gives the operation that caused the overflow and may be one of: **addition**, **unsigned addition**, **multiplication**, **unsigned multiplication**, **negation**, **shift left**, **unsigned shift left**, **subtraction**, or **unsigned sub.**

To suppress this message for particular constant operations you may have to supply explicit truncation. For example, if you want to obtain the low order 8 bits of the integer 20000 into the high byte of a 16-bit **int**, shifting left would cause this warning. However, truncating first and then shifting would be OK. The following code illustrates this where **int** is 16 bits.

```
20000u << 8;                    /* 648 */
(0xFF & 20000u) << 8;           /* OK  */
```

If you truncate with a cast you may make a signed expression out of an unsigned. For example, the following receives a warning (for 16 bit **int**).

```
(unsigned char) OxFFFu << 8       /* 648 */
```

because the **unsigned char** is promoted to **int** before shifting. The resulting quantity is actually negative. You would need to revive the **unsigned** nature of the expression with

```
(unsigned) (unsigned char) OxFFF << 8     /* OK */
```

649 **Sign fill during constant shift** -- During the evaluation of a constant expression, a negative integer was shifted right causing sign fill of vacated positions. If this is what is intended, suppress this error, but be aware that sign fill is implementation-dependent.

650 **Constant out of range for operator *'String'*** -- In a comparison operator or equality test (or implied equality test as for a **case** statement), a constant operand is not in the range specified by the other operand. For example, if 300 is compared against a **char** variable, this warning will be issued. Moreover, if **char** is signed (and 8 bits) you will get this message if you compare against an integer greater than 127. The problem can be fixed with a cast. For example:

```
if( ch == 0xFF ) ...
if( (unsigned char) ch == 0xFF ) ...
```

If **char** is signed (**+fcu** has not been set) the first receives a warning and can never succeed. The second suppresses the warning and corrects the bug.

PC-lint/FlexeLint will take into account the limited precision of some operands such as bit-fields and enumerated types. Also, PC-lint/FlexeLint will take advantage of some computations that limit the precision of an operand. For example,

```
if( (n & 0xFF) >> 4 == 16 ) ...
```

will receive this warning because the left-hand side is limited to 4 bits of precision.

651 **`Potentially confusing initializer`** -- An initializer for a complex aggregate is being processed that contains some subaggregates that are bracketed and some that are not. ANSI/ISO recommends either "minimally bracketed" initializers in which there are no interior braces or "fully bracketed" initializers in which all interior aggregates are bracketed.

652 **`#define of symbol '`*`Symbol`*`' declared previously at `*`Location`** -- A macro is being defined for a symbol that had previously been declared. For example:

```
int n;
#define n N
```

will draw this complaint. Prior symbols checked are local and global variables, functions and **`typedef`** symbols, and **`struct`**, **`union`** and **`enum`** tags. Not checked are **`struct`** and **`union`** members.

653 **`Possible loss of fraction`** -- When two integers are divided and assigned to a floating point variable the fraction portion is lost. For example, although

```
double x = 5 / 2;
```

appears to assign **`2.5`** to **`x`** it actually assigns **`2.0`**. To make sure you don't lose the fraction, cast at least one of the operands to a floating point type. If you really wish to do the truncation, cast the resulting divide to an integral (**`int`** or **`long`**) before assigning to the floating point variable.

654 **`Option `*`String`*` obsolete; use -width(W,I)`** -- The option **`-w`** is now used to set the warning level and should no longer be used to specify the width of error messages. Instead use **`-width`** with the same arguments as before to set the width. To set the warning level to 3, for example, use the option **`-w3`**, not **`-w(3)`**.

655 **`bit-wise operation uses (compatible) enum's`** -- A bit-wise operator (one of **`'|'`**, **`'&'`** or **`'^'`**) is used to combine two compatible enumerations. The type of the result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

656 **`Arithmetic operation uses (compatible) enum's`** -- An arithmetic operator (one of **`'+'`**, or **`'-'`**) is used to combine two compatible enumerations. The type of the result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

657 **`Unusual (nonportable) anonymous struct or union`** -- A **`struct`** or **`union`** declaration without a declarator was taken to be anonymous. However, the anonymous **`union`** supported by C++ and other dialects of C require untagged union's. Tagged unions and tagged or untagged structs are rarely supported, as anonymous.

658 **Anonymous union assumed (use flag +fan)** -- A **union** without a declarator was found. Was this an attempt to define an anonymous union? If so, anonymous unions should be activated with the **+fan** flag. This flag is activated automatically for C++.

659 **Nothing follows '}' on line terminating struct/union/class/enum definition** -- A **struct**/**union**/**class**/**enum** definition occurred and the closing '**}**' was not followed on the same line by another token. It looks suspicious. Missing semi-colons after such definitions can be a source of strange and mysterious messages. If you intentionally omitted the semi-colon then simply place the token, which follows, on the same line as the '**}**'. At the very least follow the '**}**' with a comment.

660 **Option '*String*' requests removing an extent that is not on the list** -- A number of options use the '**-**' prefix to remove and the '**+**' prefix to add elements to a list. For example to add (the most unusual) extension **.C++** to designate C++ processing of files bearing that extension, a programmer should employ the option:

```
+cpp(.C++)
```

However, if a leading '**-**' is employed (a natural mistake) this warning will be emitted.

661 **possible access of out-of-bounds pointer ('*Integer*' beyond end of data) by operator '*String*'** -- An out-of-bounds pointer may have been accessed. See message **415** for a description of the parameters ***Integer*** and ***String***. For example:

```
int a[10];
if( n <= 10 ) a[n] = 0;
```

Here the programmer presumably should have written **n < 10**. This message is similar to messages **415** and **796** but differs from them by the degree of probability. See Section 10.2 Value Tracking

662 **possible creation of out-of-bounds pointer ('*Integer*' beyond end of data) by operator '*String*'** -- An out-of-bounds pointer may have been created. See message **415** for a description of the parameters ***Integer*** and ***String***. For example:

```
int a[10];
if( n <= 20 ) f( a + n );
```

Here, it appears as though an illicit pointer is being created, but PC-lint/FlexeLint cannot be certain. See also messages **416** and **797**. See Section 10.2 Value Tracking

663 **Suspicious array to pointer conversion** -- This warning occurs in the following kind of situation:

```
struct x { int a; } y[2];
...  y->a  ...
```

Here, the programmer forgot to index the array but the error normally goes undetected because the array reference is automatically and implicitly converted to a pointer to the first element of the array. If you really mean to access the first element use **`y[0].a`**

664 **`Left side of logical OR (||) or logical AND (&&) does not return`** -- An exiting function was found on the left hand side of an operator implying that the right hand side would never be executed. For example:

```
if( (exit(0), n == 0) || n > 2 ) ...
```

Since the exit function does not return, control can never flow to the right hand operator.

665 **`Unparenthesized parameter`** ***`Integer`*** **`in macro '`*** ***`Symbol`*** **`' is passed an expression`** -- An expression was passed to a macro parameter that was not parenthesized. For example:

```
#define mult(a,b) (a*b)
...   mult( 100, 4 + 10 )
```

Here the programmer is beguiled into thinking that the **`4+10`** is taken as a quantity to be multiplied by **`100`** but instead results in: **`100*4+10`**, which is quite different. The recommended remedy (**[22 section 19.4]**) is to parenthesize such parameters as in:

```
#define mult(a,b) ((a)*(b))
```

The message is not arbitrarily given for any unparenthesized parameter but only when the actual macro argument sufficiently resembles an expression and the expression involves binary operators. The priority of the operator is not considered except that it must have lower priority than the unary operators. The message is not issued at the point of macro definition because it may not be appropriate to parenthesize the parameter. For example, the following macro expects that an operator will be passed as argument. It would be an error to enclose **`op`** in parentheses.

```
#define check(x,op,y) if( ((x) op (y)) == 0 ) print( ... )
```

666 **`Expression with side effects passed to repeated parameter`** ***`Integer`*** **`of macro '`*** ***`Symbol`*** **`'`** -- A repeated parameter within a macro was passed an argument with side-effects. For example:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))

... ABS( n++ )
```

Although the **ABS** macro is correctly defined to specify the absolute value of its argument, the repeated use of the parameter **x** implies a repeated evaluation of the actual argument **n++**. This results in two increments to the variable **n**. **[22 section 19.6]** Any expression containing a function call is also considered to have side-effects.

667 **Inconsistent use of qualifiers for symbol '*Symbol*' (type '*Type*' vs. '*Type*') conflicts with *Location*** -- A declaration for the identified ***Symbol*** is inconsistent with a prior declaration for the same symbol. There was a nominal difference in the declaration but owing to the memory model chosen there was no real difference. For example, in large model, one declaration declares external symbol **alpha** to be a **far** pointer and another declaration omits the memory model specification.

668 **Possibly passing a null pointer to function '*Symbol*', *Context Reference*** -- A NULL pointer is possibly being passed to a function identified by ***Symbol***. The argument in question is given by ***Context***. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option **-function** or **-sem**. See Section 11.1 Function Mimicry (-function), Section 11.2 Semantic Specifications and Section 10.2 Value Tracking.

669 **Possible data overrun for function '*Symbol*', argument *Integer* exceeds argument *Integer Reference*** -- This message is for data transfer functions such as **memcpy**, **strcpy**, **fgets**, etc. when the size indicated by the first cited argument (or arguments) can possibly exceed the size of the buffer area cited by the second. The message may also be issued for user functions via the **-function** or **-sem** option. See Section 11.1 Function Mimicry (-function), Section 11.2 Semantic Specifications and Section 10.2 Value Tracking.

670 **Possible access beyond array for function '*Symbol*', argument *Integer* exceeds *Integer Reference*** -- This message is issued for several library functions (such as **fwrite**, **memcmp**, etc.) wherein there is a possible attempt to access more data than exist. For example, if the length of data specified in the **fwrite** call exceeds the size of the data specified. The function is specified by ***Symbol*** and the arguments are identified by argument number. See also Section 11.1 Function Mimicry (-function), Section 11.2 Semantic Specifications and Section 10.2 Value Tracking.

671 **Possibly passing to function '*Symbol*' a negative value (*Integer*), *Context Reference*** -- An integral value that may possibly be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (***Symbol***), the questionable value (***Integer***) and the argument number (***Context***). The function may be a standard library function designed to accept only positive values such as **malloc** or **memcpy** (third argument), or may have been identified by the user as such through the **-function** or **-sem** options. See message **422** for an example and further explanation.

672 **Possible memory leak in assignment to pointer '*Symbol*'** -- An assignment was made to a pointer variable (designated by ***Symbol***), which may already be holding the address of an allocated object, which had not been freed. The allocation of memory, which is not freed, is considered a 'memory leak'. The memory leak is considered 'possible' because only some lines of flow will result in a leak.

673 **Possibly inappropriate deallocation (*Name1*) for '*Name2*' data** -- This message indicates that a deallocation (**free()**, **delete**, or **delete[]**) as specified by ***Name1*** may be inappropriate for the data being freed. The kind of data is one or more of: **malloc**, **new**, **new[]**, **static**, **auto**, **member**, **modified** or **constant**. The word 'Possibly' is used in the message to indicate that only some of the lines of flow to the deallocation show data inconsistent with the allocation.

674 **Returning address of auto through variable '*Symbol*'** -- The value held by a pointer variable contains the address of an **auto** variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

675 **No prior semantics associated with '*Name*' in option '*String*'** -- The -**function** option is used to transfer semantics from its first argument to subsequent arguments. However it was found that the first argument ***Name*** did not have semantics.

676 **Possibly negative subscript (*Integer*) in operator '*String*'** -- An integer whose value was possibly negative was added to an array or to a pointer to an allocated area (allocated by **malloc**, **operator new**, etc.). This message is not given for pointers whose origin is unknown since a negative subscript is in general legal.

677 **sizeof used within preprocessor statement** -- Whereas the use of **sizeof** during preprocessing is supported by a number of compilers it is not a part of the ANSI/ISO C or C++ standard.

678 **Member '*Symbol*' field length (*Integer*) too small for enum precision (*Integer*)** -- A bit field was found to be too small to support all the values of an enumeration (that was used as the base of the bit field). For example:

```
enum color { red, green, yellow, blue };
struct abc { enum color c:2; };
```

Here, the message is not given because the four enumeration values of **color** will just fit within 2 bits. However, if one additional color is inserted, Warning 678 will be issued informing the programmer of the undesirable and dangerous condition.

679 **Suspicious Truncation in arithmetic expression combining with pointer** -- This message is issued when it appears that there may have been an

unintended loss of information during an operation involving integrals before combining with a pointer whose precision is greater than the integral expression. For example:

```
//lint -sp8  pointers are 8 bytes
//lint -si4  integers are 4 bytes
char *f( char *p, int n, int m )
    {
    return p + (n + m);  // warning 679
    }
```

By the rules of C/C++, the addition **n+m** is performed independently of its context and is done at integer precision. Any overflow is ignored even though the larger precision of the pointer could easily accommodate the overflow. If, on the other hand the expression were: **p+n+m**, which parses as **(p+n)+m**, no warning would be issued.

If the expression were **p+n*m** then, to suppress the warning, a cast is needed. If **long** were the same size as pointers you could use the expression:

```
return p + ((long) n * m);
```

680 **Suspicious Truncation in arithmetic expression converted to pointer** -- An arithmetic expression was cast to pointer. Moreover, the size of the pointer is greater than the size of the expression. In computing the expression, any overflow would be lost even though the pointer type would be able to accommodate the lost information. To suppress the message, cast one of the operands to an integral type large enough to hold the pointer. Alternatively, if you are sure there is no problem you may cast the expression to an integral type before casting to pointer. See messages **647**, **776**, **790** and **679**.

681 **Loop is not entered** -- The controlling expression for a loop (either the expression within a **while** clause or the second expression within a **for** clause) evaluates initially to 0 and so it appears as though the loop is never entered.

682 **sizeof applied to a parameter '*Symbol*' whose type is a sized array** -- If a parameter is typed as an array it is silently promoted to pointer. Taking the size of such an array will actually yield the size of a pointer. Consider, for example:

```
unsigned f( char a[100] ) { return sizeof(a); }
```

Here it looks as though function **f()** will return the value 100 but it will actually return the size of a pointer, which is usually 4.

683 **function '*Symbol*' #define'd** -- This message is issued whenever the name of a function with some semantic association is defined as a macro. For example:

```
#define strlen mystrlen
```

will raise this message. The problem is that the semantics defined for **strlen** will then be lost. Consider this message an alert to transfer semantics from **strlen** to **mystrlen**, using **-function(strlen, mystrlen)**. The message will be issued for built-in functions (with built-in semantics) or for user-defined semantics. The message will not be issued if the function is defined to be a function with a similar name but with underscores either appended or prepended or both. For example:

```
#define strlen __strlen
```

will not produce this message. It will produce Info **828** instead.

684 **Passing address of auto variable '*Symbol*' into caller space** -- The address of an auto variable was passed via assignment into a location specified by the caller to the function. For example:

```
void f( int *a[] )
    {
    int n;
    a[1] = &n;
    }
```

Here the address of an auto variable (**n**) is being passed into the second element of the array passed to the function **f**. This looks suspicious because upon return, the array will contain a pointer to a variable whose lifetime is over. It is possible that this is benign since it could be that the caller to **f()** is merely passing in a working space to be discarded upon return. If this is the case, you can suppress the message for function **f()** using the option

```
-efunc(684,f)
```

See also Warning **604**.

685 **Relational operator '*String,*' always evaluates to '*String*'** -- The first ***String*** is one of '**>**', '**>=**', '**<**' or '**<=**' and identifies the relational operator. The second string is one of '**True**' or '**False**'. The message is given when an expression is compared to a constant and the precision of the expression indicates that the test will always succeed or always fail. For example,

```
char ch;
...
if( ch >= -128 ) ...
```

In this example, the precision of char **ch** is 8 bits signed (assuming the **fcu** flag has been left in the OFF state) and hence it has a range of values from -128 to 127 inclusive. Hence the test is always True.

Note that, technically, **ch** is promoted to **int** before comparing with the constant. For the purpose of this comparison we consider only the underlying precision. As another example, if **u** is an **unsigned int** then

```
if( (u & 0xFF) > 0xFF ) ...
```

will also raise message **685** because the expression on the left hand side has an effective precision of 16 bits.

686 **Option '*String*' is suspicious because of '*Name*' --** An option is considered suspicious for one of a variety of reasons. The reason is designated by a reason code that is specified by ***Name***. At this writing, the only reason code is 'unbalanced quotes'.

687 **Suspicious use of comma operator** -- A comma operator appeared unbraced and unparenthesized in a statement following an **if**, **else**, **while** or **for** clause. For example:

```
if( n > 0 ) n = 1,
n = 2;
```

Thus the comma could be mistaken for a semi-colon and hence be the source of subtle bugs.

If the statement is enclosed in curly braces or if the expression is enclosed in parentheses, the message is not issued.

688 **Cast used within preprocessor conditional statement** -- A cast was used within a preprocessor conditional statement such as **#if** or **#elif**. As an example you may have written:

```
#define VERSION 11.3
...
#if (int) VERSION == 11
...
#endif
```

Such casts are not allowed by the various C and C++ standards.

689 **Apparent end of comment ignored** -- The pair of characters '***/**' was found not within a comment. As an example:

```
void f( void*/*comment*/ );
```

This is taken to be the equivalent of:

```
void f( void*  );
```

That is, an implied blank is inserted between the '*' and the '/'. To avoid this message simply place an explicit blank between the two characters.

690 **Possible access of pointer pointing *Integer* bytes past nul character by operator '*String*'** -- Accessing past the terminating nul character is often an indication of a programmer error. For example:

```
char buf[20];
char c;
strcpy( buf, "a" );
if( i < 20 )
    c = buf[i];   // legal but suspect.
```

See also messages **448** and **836**.

691 **Suspicious use of backslash** -- The backslash character has been used in a way that may produce unexpected results. Typically this would occur within a macro such as:

```
#define A b \ // comment
```

The coder might be thinking that the macro definition will be continued on to the next line. The standard indicates, however, that the newline will not be dropped in the event of an intervening comment. This should probably be recoded as:

```
#define A b /* comment */ \
```

692 **Decimal character '*Char*' follows octal escape sequence '*String*'** -- A ***string*** was found that contains an '8' or '9' after an octal escape sequence with no more than two octal digits, e.g.

```
"\079"
```

contains two characters: Octal seven (ASCII BEL) followed by '9'. The casual reader of the code (and perhaps even the programmer) could be fooled into thinking this is a single character. If this is what the programmer intended he can also render this as

```
"\07" "9"
```

so that there can be no misunderstanding.

On the other hand,

```
"\1238"
```

will not raise a message because it is assumed that the programmer knows that octal escape sequences cannot exceed four characters (including the initial backslash).

693 **Hexadecimal digit '*Char*' immediately after '*String*' is suspicious in string literal.** -- A ***string*** was found that looks suspiciously like (but is not) a hexadecimal escape sequence; rather, it is a null character followed by letter "**x**" followed by some hexadecimal digit, e.g.:

```
"\0x62"
```

was found where the programmer probably meant to type "**\x62**". If you need precisely this sequence you can use:

```
"\0" "x62"
```

and this warning will not be issued.

694 **The type of constant '*String*' (precision *Integer*) is dialect dependent** -- A decimal integer constant that requires all the bits of an **unsigned long** for its representation has a type that depends on the dialect of C or C++ implemented by the compiler you are using. For example, the constant 3000000000 requires 32 bits for its representation. If **longs** are 32 bits, then the constant is judged to be **unsigned long** in C90, **long long** in C99 and undefined in C++.

You can remove the ambiguity by applying a clarifying suffix. If you intend this to be **unsigned** use the '**U**' suffix. If you intend this to be a **long long** use the '**LL**' suffix. If the latter and you are using C++ then turn on the **+fll** flag.

695 **Inline function '*Symbol*' defined without a storage-class specifier ('static' recommended)** -- In C99, the result of a call to a function declared with '**inline**' but not '**static**' or '**extern**' is unspecified.

Example: Let the following text represent two translation units:

```
/* In module_1.c */
void f() {}

/* In module_2.c */
inline void f() {}
void g() { f(); }  /* which f() is called? */
```

The C99 Standard dictates that the above call to **f()** from **g()** in **module_2.c** may result in the execution of either **f()**.

The programmer may avoid confusion and improve portability by using the keyword

'**static**' in addition to '**inline**'. The keyword '**extern**' can also be used along with the '**inline**' to resolve this ambiguity; however, we recommend using '**static**' because, as of this writing, more compilers correctly interpret '**static inline**'.

696 **Variable *'Symbol'* has value *'String'* that is out of range for operator *'String'*** -- The variable cited in the message is being compared (using one of the 6 comparison operations) with some other expression called the comperand. The variable has a value that is out of the range of values of this comperand. For example consider:

```
void f( unsigned char ch )
   {
   int n = 1000;
   if( ch < n )   // Message 696
   ...
```

Here a message 696 will be issued stating that **n** has a value of 1000 that is out of range because 1000 is not in the set of values that **ch** can hold (assuming default sizes of scalars).

697 **Quasi-boolean values should be equality-compared only with 0 --** A quasi-boolean value is being compared (using either **!=** or **==**) with a value that is not the literal zero. A quasi-boolean value is any value whose type is a strong boolean type and that could conceivably be something other than zero or one. This is significant because in C, all non-zero values are equally true. Example:

```
/*lint -strong(AJXb, B) */
typedef int B;
#define YES ((B)1)
#define NO ((B)0)

B f( B a, B b ) {
    B c = ( a == NO );  /* OK, no Warning here */
    B d = ( a == (b != NO) ); /* Warning 697 for == but not for != */
    B e = ( a == YES ); /* Warning 697 here */
    return d == c;      /* Warning 697 here */
  }
```

Note that if **a** and **b** had instead been declared with true boolean types, such as '**bool**' in C++ or '**_Bool**' in C99, this diagnostic would not have been issued.

698 **Casual use of realloc can create a memory leak** -- A statement of the form:

```
v = realloc( v, ... );
```

has been detected. Note the repeated use of the same variable. The problem is that **realloc** can fail to allocate the necessary storage. In so doing it will return NULL. But

then the original value of **v** is overwritten resulting in a memory leak.

19.5 C Informational Messages

701 **Shift left of signed quantity (int)** -- Shifts are normally accomplished on **unsigned** operands.

702 **Shift right of signed quantity (int)** -- Shifts are normally accomplished on **unsigned** operands. Shifting an **int** right is machine dependent (sign fill vs. zero fill).

703 **Shift left of signed quantity (long)** -- Shifts are normally accomplished on **unsigned** operands.

704 **Shift right of signed quantity (long)** -- Shifts are normally accomplished on **unsigned** operands. Shifting a **long** right is machine dependent (sign fill vs. zero fill).

705 **argument no. *Integer* nominally inconsistent with format** -- The argument to **printf** (or **fprintf** or **sprintf**) was nominally inconsistent with the format. Although the size of the quantity was appropriate, the type was similar, but not exact. (E.g., passing a **long** to a **%d** or an **int** to a **%x**) You might consider casting the quantity to the correct type. You could also suppress this message, as more flagrant violations are picked up with warnings **559** and **626**.

706 **(argument no. *Integer*) indirect object inconsistent with format** -- The type of an argument to **scanf** (or **fscanf** or **sscanf**) was inappropriate to the format. However, the argument was a pointer and it pointed to a quantity of the expected size and similar, but not expected type.

707 **Mixing narrow and wide string literals in concatenation** -- The following is an example of a mixing of narrow and wide string literals.

```
const wchar_t *s = "abc" L"def";
```

The concatenation of narrow and wide string literals results in undefined behavior for C90 and C++2003. If your compiler supports such combinations or you use a C/C++ dialect that supports such, you may either suppress this message or consider making the concatenands match.

708 **union initialization** -- There was an attempt to initialize the value of a **union**. This may not be permitted in some older C compilers. This is because of the apparent ambiguity: which member should be initialized? The standard interpretation is to apply

the initialization to the first subtype of the **union**.

712 **Loss of precision (*Context*) (*Type* to *Type*)** -- An assignment (or implied assignment, see ***Context***) is being made between two integral quantities in which the first ***Type*** is larger than the second ***Type***. A cast will suppress this message.

713 **Loss of precision (*Context*) (*Type* to *Type*)** -- An assignment (or implied assignment, see ***Context***) is being made from an unsigned quantity to a signed quantity, that will result in the possible loss of one bit of integral precision such as converting from **unsigned int** to **int**. A cast will suppress the message.

714 **Symbol '*Symbol*' (*Location*) not referenced** -- The named external variable or external function was defined but not referenced. This message is suppressed for unit checkout (-**u** option).

715 **Symbol '*Symbol*' (*Location*) not referenced** -- The named formal parameter was not referenced.

716 **while(1) ...** -- A construct of the form **while(1)** ... was found. Whereas this represents a constant in a context expecting a Boolean, it may reflect a programming policy whereby infinite loops are prefixed with this construct. Hence it is given a separate number and has been placed in the informational category. The more conventional form of infinite loop prefix is **for(;;)**

717 **do ... while(0)** -- Whereas this represents a constant in a context expecting a Boolean, this construct is probably a deliberate attempt on the part of the programmer to encapsulate a sequence of statements into a single statement, and so it is given a separate error message. **[22 section 19.7]** For example:

```
#define f(k) do {n=k; m=n+1;} while(0)
```

allows **f(k)** to be used in conditional statements as in

```
if(n>0) f(3);
 else f(2);
```

Thus, if you are doing this deliberately use **-e717**

718 **Symbol '*Symbol*' undeclared, assumed to return int** -- A function was referenced without (or before) it had been declared or defined within the current module. This is not necessarily an error and you may want to suppress such messages (See Chapter 14. LIVING WITH LINT). Note that by adding a declaration to another module, you will not suppress this message. It can only be suppressed by placing a declaration within the module being processed.

719 **Too many arguments for format (*Integer* too many)** -- The number of

arguments to a function in the **printf/scanf** family was more than what is specified in the format. This message is similar to Warning **558**, which alerts users to situations in which there were too few arguments for the format. It receives a lighter Informational classification because the additional arguments are simply ignored.

720 **Boolean test of assignment** -- An assignment was found in a context that requires a Boolean (such as in an **if()** or **while()** clause or as an operand to **&&** or **||**). This may be legitimate or it could have resulted from a mistaken use of **=** for **==**.

721 **Suspicious use of ;** -- A semi-colon was found immediately to the right of a right parenthesis in a construct of the form **if(*e*);**. As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the '**;**' is separated by at least one blank from the '**)**'. Better, place it on a separate line. See also message **548**.

722 **Suspicious use of ;** -- A semi-colon was found immediately to the right of a right parenthesis in a construct of the form **while(*e*);** or **for(*e*;*e*;*e*);** As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the '**;**' is separated by at least one blank from the '**)**'. Better, place it on a separate line.

723 **Suspicious use of =** -- A preprocessor definition began with an **=** sign. For example:

```
#define LIMIT = 50
```

Was this intentional? Or was the programmer thinking of assignment when he wrote this?

725 **Expected positive indentation from *Location*** -- The current line was found to be aligned with, rather than indented with respect to, the indicated line. The indicated line corresponds to a clause introducing a control structure and statements within its scope are expected to be indented with respect to it. If tabs within your program are other than 8 blanks you should use the **-t** option (See Section 13.3 Indentation Checking).

726 **Extraneous comma ignored** -- A comma followed by a right-brace within an enumeration is not a valid ANSI/ISO construct. The comma is ignored.

727 **Symbol '*Symbol*' (*Location*) not explicitly initialized** -- The named static variable (local to a function) was not explicitly initialized before use. The following remarks apply to messages **728** and **729** as well as **727**. By no explicit initialization we mean that there was no initializer present in the definition of the object, no direct assignment to the object, and no address operator applied to the object or, if the address of the object was taken, it was assigned to a pointer to **const**. These messages do not necessarily signal errors since the implicit initialization for static variables is 0.

However, the messages are helpful in indicating those variables that you had forgotten to initialize to a value. To extract the maximum benefit from the messages we suggest that you employ an explicit initializer for those variables that you want to initialize to 0. For example:

```
static int n = 0;
```

For variables that will be initialized dynamically, do not use an explicit initializer as in:

```
static int m;
```

This message will be given for any array, **struct** or **union** if no member or element has been assigned a value.

728 **Symbol '*Symbol*' (*Location*) not explicitly initialized** -- The named intra-module variable (static variable with file scope) was not explicitly initialized. See the comments on message **727** for more details.

729 **Symbol '*Symbol*' (*Location*) not explicitly initialized** -- The named inter-module variable (external variable) was not explicitly initialized. See the comments on message **727** for more details. This message is suppressed for unit checkout (**-u**).

730 **Boolean argument to function** -- A Boolean was used as an argument to a function. Was this intended? Or was the programmer confused by a particularly complex conditional statement? Experienced C programmers often suppress this message. This message is given only if the associated parameter is not declared bool.

731 **Boolean argument to equal/not equal** -- A Boolean was used as an argument to **==** or **!=**. For example:

```
if( (a > b) == (c > d) ) ...
```

tests to see if the inequalities are of the same value. This could be an error as it is an unusual use of a Boolean (see Warnings **503** and **514**) but it may also be deliberate since this is the only way to efficiently achieve equivalence or exclusive or. Because of this possible use, the construct is given a relatively mild 'informational' classification. If the Boolean argument is cast to some type, this message is not given.

732 **Loss of sign (*Context*) (*Type* to *Type*)** -- An assignment (or implied assignment, see ***Context***) is made from a signed quantity to an unsigned quantity. Also, it could not be determined that the signed quantity had no sign. For example:

```
u  =  n;      /* Info 732 */
u  =  4;      /*  OK      */
```

where **u** is unsigned and **n** is not, warrants a message only for the first assignment, even though the constant **4** is nominally a signed **int**.

Make sure that this is not an error (that the assigned value is never negative) and then use a cast (to unsigned) to remove the message.

733 **Assigning address of auto variable '*Symbol*' to outer scope symbol '*Symbol*'** -- The address of an **auto** variable is only valid within the block in which the variable is declared. An address to such a variable has been assigned to a variable that has a longer life expectancy. There is an inherent danger in doing this.

734 **Loss of precision (*Context*) (*Integer* bits to *Integer* bits)** -- An assignment is being made into an object smaller than an **int**. The information being assigned is derived from another object or combination of objects in such a way that information could potentially be lost. The number of bits given does not count the sign bit. For example if **ch** is a **char** and **n** is an **int** then:

```
ch  =  n;
```

will trigger this message whereas:

```
ch  =  n & 1;
```

will not. To suppress the message a cast can be made as in:

```
ch  =  (char) n;
```

You may receive notices involving multiplication and shift operators with subinteger variables. For example:

```
ch = ch << 2
ch = ch * ch
```

where, for example, **ch** is an **unsigned char**. These can be suppressed by using the flag **+fpm** (precision of an operator is bound by the maximum of its operands). See Section 5.5 Flag Options

735 **Loss of precision (*Context*) (*Integer* bits to *Integer* bits)** -- An assignment (or implied assignment, see ***Context***) is made from a **long double** to a **double**. Using a cast will suppress the message. The number of bits includes the sign bit.

736 **Loss of precision (*Context*) (*Integer* bits to *Integer* bits)** -- An assignment (or implied assignment, see ***Context***) is being made to a float from a value or combination of values that appear to have higher precision than a float. You may suppress this message by using a cast. The number of bits includes the sign bit.

737 **Loss of sign in promotion from *Type* to *Type*** -- An unsigned quantity was joined with a signed quantity in a binary operator (or 2nd and 3rd arguments to the conditional operator **? :**) and the signed quantity is implicitly converted to unsigned. The message will not be given if the signed quantity is an unsigned constant, a Boolean, or an expression involving bit manipulation. For example,

```
u  &  ~0xFF
```

where **u** is unsigned does not draw the message even though the operand on the right is technically a signed integer constant. It looks enough like an unsigned to warrant not giving the message.

This mixed mode operation could also draw Warnings **573** or **574** depending upon the operator involved.

You may suppress the message with a cast but you should first determine whether the signed value could ever be negative or whether the unsigned value can fit within the constraints of a signed quantity.

738 **Symbol '*Symbol*' (*Location*) not explicitly initialized** -- The named **static** local variable was not initialized before being passed to a function whose corresponding parameter is declared as pointer to **const**. Is this an error or is the programmer relying on the default initialization of 0 for all static items? By employing an explicit initializer you will suppress this message. See also message numbers **727** and **603**.

739 **Trigraph Sequence '*String*' in literal (Quiet Change)** -- The indicated Trigraph (three-character) sequence was found within a string. This trigraph reduces to a single character according to the ANSI/ISO standards. This represents a "Quiet Change" from the past where the sequence was not treated as exceptional. If you had no intention of mapping these characters into a single character you may precede the initial **'?'** with a backslash. If you are aware of the convention and you intend that the Trigraph be converted you should suppress this informational message.

740 **Unusual pointer cast (incompatible indirect types)** -- A cast is being made to convert one pointer to another such that neither of the pointers is a generic pointer (neither is pointer to **char**, **unsigned char**, or **void**) and the indirect types are truly different. The message will not be given if the indirect types differ merely in signedness (e.g., pointer to **unsigned** versus pointer to **int**) or in qualification (e.g., pointer to **const int** versus pointer to **int**). The message will also not be given if one of the indirect types is a **union**.

The main purpose of this message is to report possible problems for machines in which pointer to **char** is rendered differently from pointer to word. Consider casting a pointer to pointer to **char** to a pointer to pointer to word. The indirect bit pattern remains

unchanged.

A second reason is to identify those pointer casts in which the indirect type doesn't seem to have the proper bit pattern such as casting from a pointer to **int** to a pointer to **double**.

If you are not interested in running on machines in which **char** pointers are fundamentally different from other pointers then you may want to suppress this message. You can also suppress this message by first casting to **char** pointer or to **void** pointer but this is only recommended if the underlying semantics are right.

741 **Unusual pointer cast (function qualification)** -- A cast is being made between two pointers such that their indirect types differ in one of the Microsoft qualifiers: **pascal**, **fortran**, **cdecl** and **interrupt**. If this is not an error, you may cast to a more neutral pointer first such as a **void ***.

742 **Multiple character constant** -- A character constant was found that contained multiple characters, e.g., **'ab'**. This is legal C but the numeric value of the constant is implementation defined. It may be safe to suppress this message because, if more characters are provided than what can fit in an **int**, message number **25** is given.

743 **Negative character constant** -- A character constant was specified whose value is some negative integer. For example, on machines where a byte is 8 bits, the character constant **'\xFF'** is flagged because its value (according to the ANSI/ISO standard) is **-1** (its type is **int**). Note that its value is not **0xFF**.

744 **switch statement has no default** -- A **switch** statement has no section labeled **default:**. Was this an oversight? It is standard practice in many programming groups to always have a **default:** case. This can lead to better (and earlier) error detection. One way to suppress this message is by introducing a vacuous **default: break;** statement. If you think this adds too much overhead to your program, think again. In all cases tested so far, the introduction of this statement added absolutely nothing to the overall length of code. If you accompany the vacuous statement with a suitable comment, your code will at least be more readable.

This message is not given if the control expression is an enumerated type. In this case, all enumerated constants are expected to be represented by **case** statements, else **787** will be issued.

745 **function '*Name*' has no explicit type or class, int assumed** -- A function declaration or definition contained no explicit type. Was this deliberate? If the flag **fdr** (deduce return mode, see Section 5.5 Flag Options) is turned on, this message is suppressed.

746 **call to function '*Name*' not made in the presence of a prototype** -- A call to a function is not made in the presence of a prototype. This does not mean

that PC-lint/FlexeLint is unaware of any prototype; it means that a prototype is not in a position for a compiler to see it. If you have not adopted a strict prototyping convention you will want to suppress this message with **-e746**.

747 **Significant prototype coercion (*Context*) *Type* to *Type*** -- The type specified in the prototype differed from the type provided as an argument in some significant way. Usually the two types are arithmetic of differing sizes or one is **float** and the other integral. This is flagged because if the program were to be translated by a compiler that does not support prototype conversion, the conversion would not be performed. See also Elective Notes **917** and **918.**

748 **Symbol '*Symbol*' (*Location*) is a register variable used with setjmp** -- The named variable is a register variable and is used within a function that calls upon **setjmp**. When a subsequent **longjmp** is issued, the values of register variables may be unpredictable. If this error is not suppressed for this variable, the variable is marked as uninitialized at this point in the program.

More information on messages 749-769 can be found in Section 13.8 Weak Definials.

749 **local enumeration constant '*Symbol*' (*Location*) not referenced** -- A member (name provided as ***Symbol***) of an **enum** was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. Compare with messages **754** and **769**.

750 **local macro '*Symbol*' (*Location*) not referenced** -- A 'local' macro is one that is not defined in a header file. The macro is not referenced throughout the module in which it is defined.

751 **local typedef '*Symbol*' (*Location*) not referenced** -- A 'local' **typedef** symbol is one that is not defined in any header file. It may have file scope or block scope but it was not used through its scope.

752 **local declarator '*Symbol*' (*Location*) not referenced** -- A 'local' declarator symbol is one declared in a declaration appearing in the module file itself as opposed to a header file. The symbol may have file scope or may have block scope. But it wasn't referenced.

753 **local struct, union or enum tag '*Symbol*' (*Location*) not referenced** -- A 'local' tag is one not defined in a header file. Since its definition appeared, why was it not used? Use of a tag is implied by the use of any of its members.

754 **local structure member '*Symbol*' (*Location*) not referenced** -- A member (name provided as ***Symbol***) of a **struct** or **union** was defined in a module but was not otherwise used within that module. A 'local' member is one that is not

defined in a header file. See message **768**.

755 **global macro '*Symbol*' (*Location*) not referenced** -- A 'global' macro is one defined in a header file. This message is given for macros defined in non-library headers. The macro is not used in any of the modules comprising the program. This message is suppressed for unit checkout (**-u** option). See Section 13.8 Weak Definials

756 **global typedef '*Symbol*' (*Location*) not referenced** -- This message is given for a **typedef** symbol declared in a non-library header file. The symbol is not used in any of the modules comprising a program. This message is suppressed for unit checkout (**-u** option).

757 **global declarator '*Symbol*' (*Location*) not referenced** -- This message is given for objects that have been declared in non-library header files and that have not been used in any module comprising the program being checked. The message is suppressed for unit checkout (**-u**).

758 **global struct, union or enum tag '*Symbol*' (*Location*) not referenced** -- This message is given for **struct**, **union** and **enum** tags that have been defined in non-library header files and that have not been used in any module comprising the program. The message is suppressed for unit checkout (**-u**).

759 **header declaration for symbol '*Symbol*' (*Location*) could be moved from header to module** -- This message is given for declarations, within non-library header files, that are not referenced outside the defining module. Hence, it can be moved inside the module and thereby 'lighten the load' on all modules using the header. This message is only given when more than one module is being linted.

760 **Redundant macro '*Symbol*' defined identically at *Location*** -- The given macro was defined earlier (location given) in the same way and is hence redundant.

761 **Redundant typedef '*Symbol*' previously declared at *Location*** -- A **typedef** symbol has been **typedef** earlier at the given location. Although the declarations are consistent you should probably remove the second.

762 **Redundantly declared symbol '*Symbol*' previously declared at *Location*** -- A declaration for the given symbol was found to be consistent with an earlier declaration in the same scope. This declaration adds nothing new and it can be removed.

763 **Redundant declaration for symbol '*Symbol*' previously declared at *Location*** -- A tag for a **struct**, **union** or **enum** was defined twice in the same module (consistently). The second one can be removed.

764 **switch statement does not have a case** -- A switch statement has been found that does not have a **case** statement associated with it (it may or may not have a **default** statement). This is normally a useless construct.

765 **external '*Symbol*' (*Location*) could be made static** -- An external symbol was referenced in only one module. It was not declared **static** (and its type is not qualified with the Microsoft keyword __**export**). Some programmers like to make **static** every symbol they can, because this lightens the load on the linker. It also represents good documentation. On the other hand, you may want the symbol to remain external because debuggers often work only on external names. It's possible, using macros, to have the best of both worlds; see Section 13.8.3 static-able.

766 **Include of header file *'FileName'* not used in module *'FileName'*** -- The named header file was directly included in the named module but the **#include** can be removed because it was not used in processing the named module or in any header included by the module. It contained no macro, **typedef**, **struct**, **union** or **enum** tag or component, or declaration referenced by the module. One of the reasons a particular **#include** can be removed is because it had been included by an earlier header file. Warning **537** can be used to detect such cases. Note: Through conditional compilation some seeming anomalies could occur so that header files that are reported as not used might be regarded as used if they are placed earlier in a sequence of includes. See also Elective Notes **964**, **966** and Section 13.8.1 Unused Headers.

767 **macro '*Symbol*' was defined differently in another module (*Location*)** -- Two macros processed in two different modules had inconsistent definitions.

768 **global struct member '*Symbol*' (*Location*) not referenced** -- A member (name provided as ***Symbol***) of a **struct** or **union** appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. Since **struct**'s may be replicated in storage, finding an unused member can pay handsome storage dividends. However, many structures merely reflect an agreed upon convention for accessing storage and for any one program many members are unused. In this case, receiving this message can be a nuisance. One convenient way to avoid unwanted messages (other than the usual **-e** and **-esym**) is to always place such structures in library header files. Alternatively, you can place the **struct** within a **++flb ... --flb** sandwich to force it to be considered library.

769 **global enumeration constant '*Symbol*' (*Location*) not referenced** -- A member (name provided as ***Symbol***) of an **enum** appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. There are reasons why a programmer may occasionally want to retain an unused **enum** and for this reason this message is distinguished from **768** (unused member). See message **768** for ways of selectively suppressing this message.

770 **tag '*Symbol*' defined identically at *Location*** -- The **struct**, **union**, or **enum** tag ***Symbol*** was defined identically in different locations (usually two different files). This is not an error but it is not necessarily good programming practice either. It is better to place common definitions of this kind in a header file where they can be shared among several modules. If you do this, you will not get this message. Note that if the tag is defined differently in different scopes, you will receive warning **631** rather than this message.

771 **Symbol '*Symbol*' (*Location*) conceivably not initialized** -- The named symbol, declared at ***Location***, was initialized in the main portion of a control loop (**while**, **for**, or **do**) and subsequently used outside the loop. If it is possible for the main body of the loop to not be fully executed, then the given symbol would remain uninitialized resulting in an error.

PC-lint/FlexeLint does not do a great job of evaluating expressions and hence may not recognize that a loop is executed at least once. This is particularly true after initializing an array. Satisfy yourself that the loop is executed and then suppress the message. You may wish to suppress the message globally with **-e771** or just for specific symbols using **-esym**. Don't forget that a simple assignment statement may be all that's needed to suppress the message. See Section 10.1 Initialization Tracking

772 **Symbol '*Symbol*' (*Location*) conceivably not initialized** -- The address of the named ***Symbol*** was passed to a function expecting to receive a pointer to a **const** item. This requires the ***Symbol*** to have been initialized. See Warning **603** for an explanation of the dangers of such a construct. See Informational message **771** for an explanation of "conceivably not initialized."

773 **Expression-like macro '*Symbol*' not parenthesized** -- A macro that appeared to be an expression contained unparenthesized binary operators and therefore may result in unexpected associations when used with other operators. For example,

```
#define A  B + 1
```

may be used later in the context:

```
f( A * 2 );
```

with the surprising result that **B+2** gets passed to **f** and not the **(B+1)*2** Corrective action is to define **A** as:

```
#define A  (B + 1)
```

Highest precedence binary operators are not reported upon. Thus:

```
#define A  s.x
```

does not elicit this message because this case does not seem to represent a problem. Also, unparenthesized unary operators (including casts) do not generate this message. Information about such unparenthesized parameters can be found by enabling Elective Note **973**. **[22 section 19.5]**

774 **Boolean within '*String*' always evaluates to [True/False]** -- The indicated clause (***String*** is one of **if**, **while** or **for** (2nd expression)) has an argument that appears to always evaluate to either '**True**' or '**False**' (as indicated in the message). Information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message **506**, which is based on testing constants or combinations of constants. Also compare with the Elective Note **944**, which can sometimes provide more detailed information. See Section 10.2 Value Tracking

775 **non-negative quantity cannot be less than zero** -- A non-negative quantity is being compared for being **<=0**. This is a little suspicious since a non-negative quantity can be equal to **0** but never less than **0**. The non-negative quantity may be of type **unsigned** or may have been promoted from an **unsigned** type or may have been judged not to have a sign by virtue of it having been AND'ed with a quantity known not to have a sign bit, an **enum** that may not be negative, etc. See also Warning **568**.

776 **Possible truncation of addition** -- An **int** expression (signed or unsigned) involving addition or subtraction is converted to **long** implicitly or explicitly. Moreover, the precision of a **long** is greater than that of **int**. If an overflow occurred, information would be lost. Either cast one of the operands to some form of **long** or cast the result to some form of **int**.

See Warning **647** for a further description and an example of this kind of error. See also messages **790** and **942**.

777 **Testing float's for equality** -- This message is issued when the operands of operators **==** and **!=** are some form of floating type (**float**, **double**, or **long double**). Testing for equality between two floating point quantities is suspect because of round-off error and the lack of perfect representation of fractions. If your numerical algorithm calls for such testing turn the message off. The message is suppressed when one of the operands can be represented exactly, such as 0 or 13.5.

778 **Constant expression evaluates to 0 in operation: '*String*'** -- A constant expression involving addition, subtraction, multiplication, shifting, or negation resulted in a 0. This could be a purposeful computation but could also have been unintended. If this is intentional, suppress the message. If one of the operands is 0, Elective Note **941** may be issued rather than a 778.

779 **String constant in comparison operator: *Operator*** -- A string constant appeared as an argument to a comparison operator. For example:

```
if( s == "abc" ) ...
```

This is usually an error. Did the programmer intend to use **strcmp**? It certainly looks suspicious. At the very least, any such comparison is bound to be machine-dependent. If you cast the string constant, the message is suppressed.

780 **Vacuous array element** -- A declaration of an array looks suspicious because the array element is an array of 0 dimension. For example:

```
extern int a[][];
extern int a[10][];
```

will both emit this message but

```
extern int a[][10];
```

will not. In the latter case, proper array accessing will take place even though the outermost dimension is missing.

If **extern** were omitted, the construct would be given a more serious error message.

782 **Line exceeds *Integer* characters** -- An internal limit on the size of the input buffer has been reached. The message contains the maximum permissible size. This does not necessarily mean that the input will be processed erroneously. Additional characters will be read on a subsequent read. However the line sequence numbers reported on messages will be incorrect.

783 **Line does not end with new-line** -- This message is issued when an input line is not terminated by a new-line or when a NUL character appears within an input line. When input lines are read, an **fgets** is used. A **strlen** call is made to determine the number of characters read. If the new-line character is not seen at the presumed end, this message is issued. If your editor is in the habit of not appending new-lines onto the end of the last line of the file then suppress this message. Otherwise, examine the file for NUL characters and eliminate them.

784 **Nul character truncated from string** -- During initialization of an array with a string constant there was not enough room to hold the trailing NUL character. For example:

```
char a[3] = "abc";
```

would evoke such a message. This may not be an error since the easiest way to do this initialization is in the manner indicated. It is more convenient than:

```
char a[3] = { 'a', 'b', 'c' };
```

On the other hand, if it really is an error it may be especially difficult to find.

785 **`Too few initializers for aggregate`** -- The number of initializers in a brace-enclosed initializer was less than the number of items in the aggregate. Default initialization is taken. An exception is made with the initializer **`{0}`**. This is given a separate message number in the Elective Note category (**943**). It is normally considered to be simply a stylized way of initializing all members to 0.

786 **`String concatenation within initializer`** -- Although it is perfectly 'legal' to concatenate string constants within an initializer, this is a frequent source of error. Consider:

```
char *s[] = { "abc" "def" };
```

Did the programmer intend to have an array of two strings but forget the comma separator? Or was a single string intended?

787 **`enum constant '`*`Symbol`*`' not used within switch`** -- A **`switch`** expression is an enumerated type and at least one of the enumerated constants was not present as a **`case`** label. Moreover, no **`default`** case was provided.

788 **`enum constant '`*`Symbol`*`' not used within defaulted switch`** -- A **`switch`** expression is an enumerated type and at least one of the enumerated constants was not present as a **`case`** label. However, unlike Info **787**, a **`default`** case was provided. This is a mild form of the case reported by Info **787**. The user may thus elect to inhibit this mild form while retaining Info **787**.

789 **`Assigning address of auto variable '`*`Symbol`*`' to static`** -- The address of an **`auto`** variable (***`Symbol`***) is being assigned to a **`static`** variable. This is dangerous because the **`static`** variable will persist after return from the function in which the **`auto`** is declared but the **`auto`** will be, in theory, gone. This can prove to be among the hardest bugs to find. If you have one of these, make certain there is no error and use **-`esym`** to suppress the message for a particular variable.

790 **`Suspicious truncation, integral to float`** -- This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals, the result of which is later converted to a floating point quantity. The operations that are scrutinized and reported upon by this message are: shift left and multiplication. Addition and subtraction are covered by Elective Note **942**. See also messages **647** and **776**.

791 **`unusual option sequence`** -- A temporary message suppression option (one having the form: **`!e...`**) followed a regular option. Was this intended?

792 **`void cast of void expression`** -- A **`void`** expression has been cast to **`void`**. Was this intended?

793 **`ANSI/ISO limit of String 'Name' exceeded`** -- Some ANSI/ISO limit has been exceeded. These limits are described under the heading "Translation limits" the ANSI/ISO C Standards **[1] [4]** and under the heading "Implementation Quantities" in the C++ standards **[10] [34]**. Programs exceeding these limits are not considered maximally portable. However, they may work for individual compilers.

Say a large program exceeds the ANSI/ISO limit of 4095 external identifiers. This will result in message **`793`** "**`ANSI limit of 4059 'external identifiers' exceeded`**". It may not be obvious how to inhibit this message for identifiers while leaving other limits in a reportable state. The second parameter of the message is designated ***`Name`*** and so the **`-esym`** may be used. Because the symbol contains a blank, quotes must be used. The option becomes:

```
-"esym(793,external identifiers)"
```

794 **`Conceivable use of null pointer 'Symbol' in [left/right] argument to operator 'String' Reference`** -- From information gleaned from earlier statements it is conceivable that a null pointer (a pointer whose value is 0) can be used in a context where null pointers are inappropriate. In the case of binary operators, one of the words '**`left`**' or '**`right`**' is used to designate which operand is null. ***`Symbol`*** identifies the pointer variable that may be NULL. This is similar to messages **`413`** and **`613`** and differs from them in that the likelihood is not as great. For example:

```
int *p = 0;
int i;
for( i = 0; i < n; i++ )
    p = &a[i];
*p = 0;
```

If the body of the **`for`** loop is never taken, then **`p`** remains null.

795 **`Conceivable division by 0`** -- In a division or modulus operation the division is deduced to be conceivably 0. See Section 10.2 Value Tracking for the meaning of "conceivable".

796 **`Conceivable access of out-of-bounds pointer ('Integer' beyond end of data) by operator 'String'`** -- An out-of-bounds pointer may conceivably have been accessed. See message **`415`** for a description of the parameters ***`Integer`*** and ***`String`***. For example:

```
int a[10];
int j = 100;
for( i = 0; i < n; i++ )
```

```
    j = n;
a[j] = 0;
```

Here, the access to **a[j]** is flagged because it is conceivable that the **for** loop is not executed leaving the unacceptable index of 100 in variable **j**. This message is similar to messages **415** and **661** but differing from them by the degree of probability.

797 **Conceivable creation of out-of-bounds pointer ('*Integer*' beyond end of data) by operator '*String*'** -- An out-of-bounds pointer is potentially being created. See message **415** for a description of the parameters ***Integer*** and ***String***. See message **796** for an example of how a probability can be considered 'conceivable'. See also message **416** and/or Section 10.2 Value Tracking for a description of the difference between pointer 'creation' and pointer 'access'.

798 **Redundant character '*Char*'** -- The indicated character ***char*** is redundant and can be eliminated from the input source. A typical example is a backslash on a line by itself.

799 **numerical constant '*Integer*' larger than unsigned long** -- An integral constant was found to be larger than the largest value allowed for **unsigned long** quantities. By default, an **unsigned long** is 4 bytes but can be respecified via the option **-sl#**. If the **long long** type is permitted (see option **+fll**) this message is automatically suppressed. See also message **417**.

801 **Use of goto is deprecated** -- A **goto** was detected. Use of the **goto** is not considered good programming practice by most authors and its use is normally discouraged. There are a few cases where the **goto** can be effectively employed but often these can be rewritten just as effectively without the **goto**. The use of **goto** statements can have a devastating effect on the structure of large functions creating a mass of spaghetti-like confusion. For this reason its use has been banned in many venues.

802 **Conceivably passing a null pointer to function '*Symbol*', *Context Reference*** -- A NULL pointer is conceivably being passed to a function identified by ***Symbol***. The argument in question is given by ***Context***. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option **-function** or **-sem**. See Section 11.1 Function Mimicry (-function) See Section 10.2 Value Tracking for the meaning of 'conceivable'.

803 **Conceivable data overrun for function '*Symbol*', argument *Integer* exceeds argument *Integer Reference*** -- This message is for data transfer functions such as **memcpy**, **strcpy**, **fgets**, etc. when the size indicated by the first cited argument (or arguments) can conceivably exceed the size of the buffer area cited by the second. The message may also be issued for user functions via the **-function** option or **-sem**. See Section 11.1 Function Mimicry (-function) See Section 10.2 Value Tracking for the meaning of 'conceivable'.

804 **Conceivable access beyond array for function '*Symbol*', argument *Integer* exceeds *Integer Reference*** -- This message is issued for several library functions (such as **fwrite**, **memcmp**, etc.) wherein there is conceivably an attempt to access more data than exist. For example, if the length of data specified in the **fwrite** call can exceed the size of the data specified. The function is specified by ***Symbol*** and the arguments are identified by argument number. See also Section 11.1 Function Mimicry (-function) and Section 10.2 Value Tracking.

805 **Expected L"..." to initialize wide char string** -- An initializer for a wide character array or pointer did not use a preceding '**L**'. For example:

```
wchar_t a[] = "abc";
```

was found whereas

```
wchar_t a[] = L"abc":
```

was expected.

806 **Small bit field is signed rather than unsigned** -- A small bit field (less than an **int** wide) was found and the base type is signed rather than unsigned. Since the most significant bit is a sign bit this practice can produce surprising results. For example,

```
struct { int b:1; } s;
s.b = 1;
if( s.b > 0 ) /* should succeed but actually fails */
    ...
```

807 **Conceivably passing to function '*Symbol*' a negative value (*Integer*), *Context Reference*** -- An integral value that may conceivably be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (***Symbol***), the questionable value (***Integer***) and the argument number (***Context***). The function may be a standard library function designed to accept only positive values such as **malloc** or **memcpy** (third argument), or may have been identified by the user as such through the **-function** or **-sem** options. See message **422** for an example and further explanation.

808 **No explicit type given symbol '*Symbol*', int assumed** -- An explicit type was missing in a declaration. Unlike Warning **601**, the declaration may have been accompanied by a storage class or modifier (qualifier) or both. For example:

```
extern f(void);
```

will draw message 808. Had the **extern** not been present, a **745** would have been raised.

The keywords **unsigned**, **signed**, **short** and **long** are taken to be explicit type specifiers even though **int** is implicitly assumed as a base.

809 **Possible return of address of auto through variable '*Symbol*'** -- The value held by a pointer variable may have been the address of an **auto** variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

810 **Arithmetic modification of custodial variable '*Symbol*'** -- We define the custodial variable as that variable directly receiving the result of a **malloc** or **new** or equivalent call. It is inappropriate to modify such a variable because it must ultimately be **free**'ed or **delete**'ed. You should first make a copy of the custodial pointer and then modify the copy. The copy is known as an alias.

811 **Possible deallocation of pointer alias** -- A **free** or a **delete** was applied to a pointer that did not appear to be the custodial variable of the storage that had been allocated. Please refer to message **810** for the definition of 'custodial variable'. Deleting an alias pointer is bad because it can result in deleting the same area twice. This can cause strange behavior at unpredictable times. Always try to identify the custodial pointer as opposed to copies (or aliases) of it. Then deallocate storage through the custodial pointer. Modify only the alias pointers.

812 **static variable '*Symbol*' has size '*Integer*'** -- The amount of storage for a static symbol has reached or exceeded a value that was specified in a **-size** option (See Section 5.7 Other Options).

813 **auto variable '*Symbol*' in function '*Symbol*' has size '*Integer*'** -- The amount of storage for an auto symbol has reached or exceeded a value that was specified in a **-size** option (See Section 5.7 Other Options).

814 **useless declaration** -- A tagless **struc**t was declared without a declarator. For example:

```
struct { int n; };
```

Such a declaration cannot very well be used.

815 **Arithmetic modification of unsaved pointer** -- An allocation expression (**malloc**, **calloc**, **new**) is not immediately assigned to a variable but is used as an operand in some expression. This would make it difficult to free the allocated storage. For example:

```
p = new X[n] + 2;
```

will elicit this message. A preferred sequence is:

```
q = new X[n];
p = q+2;
```

In this way the storage may be freed via the custodial pointer **q**.

Another example of a statement that will yield this message is:

```
p = new (char *) [n];
```

This is a gruesome blunder on the part of the programmer. It does NOT allocate an array of pointers as a novice might think. It is parsed as:

```
p = (new (char *)) [n];
```

which represents an allocation of a single pointer followed by an index into this 'array' of one pointer.

816 **Non-ANSI format specification** -- A non-standard format specifier was found in a format-processing function such as **printf** or **scanf**. Such a specifier could be unique to a particular compiler or could be a de facto standard but is not standard. The format was recognized as being a common extension. If the format was not recognized a more severe warning (**557**) would have been issued. The formats that are recognized are the **printf** codes **%c** (for wide characters) and **%s** (for string of wide characters). If you are using these for this purpose just disable the message.

817 **Conceivably negative subscript (*Integer*) in operator '*String*'** -- An integer whose value was conceivably negative was added to an array or to a pointer to an allocated area (allocated by **malloc**, **operator new**, etc.). This message is not given for pointers whose origin is unknown since a negative subscript is in general legal.

The addition could have occurred as part of a subscript operation or as part of a pointer arithmetic operation. The operator is denoted by ***String***. The value of the integer is given by ***Integer***.

818 **Pointer parameter '*Symbol*' (*Location*) could be declared ptr to const** -- As an example:

```
int f( int *p ) { return *p; }
```

can be redeclared as:

```
int f( const int *p ) { return *p; }
```

Declaring a parameter a pointer to **const** offers advantages that a mere pointer does not. In particular, you can pass to such a parameter the address of a **const** data item. In addition it can offer better documentation.

Other situations in which a **const** can be added to a declaration are covered in messages **952**, **953**, **954** and **1764**.

820 **Boolean test of a parenthesized assignment** -- A Boolean test was made on the result of an assignment and, moreover, the assignment was parenthesized. For example:

```
if ( (a = b) ) ...  // Info 820
```

will draw this informational whereas

```
if ( a = b ) ...    // Info 720
```

(i.e. the unparenthesized case) will, instead, draw Info **720**. We, of course, do not count the outer parentheses, required by the language, that always accompany the **if** clause.

The reason for partitioning the messages in this fashion is to allow the programmer to adopt the convention, advanced by some compilers (in particular **gcc**), of always placing a redundant set of parentheses around any assignment that is to be tested. In this case you can suppress Info 820 (via **-e820**) while still enabling Info **720**.

821 **Right hand side of assignment not parenthesized** -- An assignment operator was found having one of the following forms:

```
a = b || c
a = b && c
a = b ? c : d
```

Moreover, the assignment appeared in a context where a value was being obtained. For example:

```
f( a = b ? c : d );
```

The reader of such code could easily confuse the assignment for a test for equality. To eliminate any such doubts we suggest parenthesizing the right hand side as in:

```
f( a = (b ? c : d) );
```

825 **control flows into case/default without -fallthrough comment** -- A common programming mistake is to forget a break statement between case statements of a switch. For example:

```
case 'a':  a = 0;
case 'b':  a++;
```

Is the fall through deliberate or is this a bug? To signal that this is intentional use the **-fallthrough** option within a lint comment as in:

```
case 'a':  a = 0;
    //lint -fallthrough
case 'b':  a++;
```

This message is similar to Warning **616** ("control flows into case/default") and is intended to provide a stricter alternative. Warning **616** is suppressed by any comment appearing at the point of the fallthrough. Thus, an accidental omission of a break can go undetected by the insertion of a neutral comment. This can be hazardous to well-commented programs.

826 **Suspicious pointer-to-pointer conversion (area too small)** -- A pointer was converted into another either implicitly or explicitly. The area pointed to by the destination pointer is larger than the area that was designated by the source pointer. For example:

```
long *f( char *p ) { return (long *) p; }
```

827 **Loop not reachable** -- A loop structure (**for**, **while**, or **do**) could not be reached. Was this an oversight? It may be that the body of the loop has a labeled statement and that the plan of the programmer is to jump into the middle of the loop through that label. It is for this reason that we give an Informational message and not the Warning (**527**) that we would normally deliver for an unreachable statement. But please note that jumping into a loop is a questionable practice in any regard.

828 **Semantics of function '*Name*' copied to function '*Name*'** -- A function with built-in semantics or user-defined semantics was **#define**'d to be some other function with a similar name formed by prepending or appending underscores. For example:

```
#define strcmp(a,b) __strcmp__(a,b)
```

will cause Info **828** to be issued. As the message indicates, the semantics will be automatically transferred to the new function.

829 **A +headerwarn option was previously issued for header '*Symbol*'** -- Some coding standards discourage or even prohibit the use of certain header files. PC-lint/FlexeLint can guard against their use by activating the lint option **+headerwarn**(***Symbol***). Later, if the file is used, we will then issue this message.

830 **Location cited in prior message** -- Message **830** is a vehicle to convey in

'canonical form' the location information embedded within some other message. For example, consider the (somewhat simplified) message:

```
file x.c line 37:  Declaration for 'x' conflicts with line 22
```

This contains the location ("line 22") embedded in the text of the message. Embedded location information is not normally understood by editors and IDE's (Interactive Development Environments) which can only position to the nominal location (line 37 in this example). By adding this additional message with the nominal location of line 22 the user can, by stepping to the next message and, in this case, see what the 'conflict' is all about. This message and message **831** below do not follow the ordinary rules for message suppression. If they did then when the option **-w2** was employed to turn the warning level down to 2, these messages (at level 3) would also vanish. Instead they continue to function as expected. To inhibit them you need to explicitly turn them off using one of:

```
-e830
-e831
```

They may be restored via **+e830** and **+e831**; the state of suppression can be saved and restored via the **-save -restore** options. Options such as **-e8*** and **-e{831}** will have no effect on messages **830** and **831**.

831 **Reference cited in prior message** -- Message **831** is similar to message **830** in that it is a vehicle to convey in 'canonical form' location information embedded within some other message. In the case of Info **831** the information is 'Reference' information. This is a sequence of 1 or more locations that support a particular message. For example, consider the (somewhat simplified) message:

```
file y.c line 701:  Possible divide by 0
[Reference: file z.c lines 22, 23]
```

Accompanying this message will be two Info **831** messages, one for each of the references cited in the message. Without this it would be a relatively tedious matter to locate each one of the references to determine just why there is a potential divide by 0. With these additional messages, editors and IDE's can automatically position the focus of editing to the nominal locations of the message.

832 **Parameter '*Symbol*' not explicitly declared, int assumed** -- In an old-style function definition a parameter was not explicitly declared. To illustrate:

```
void f( n, m )
    int n;
    { ...
```

This is an example of an old-style function definition with **n** and **m** the parameters. **n** is

explicitly declared and `m` is allowed to default to **`int`**. An 832 will be issued for `m`.

833 **`Symbol '`*`Symbol`*`' is typed differently (`*`String`*`) in another module, compare with`*`Location`* -- Two objects, functions or definials are typed differently in two different modules. This is a case where the difference is legal but may cause confusion on the part of program maintenance.

834 **`Operator '`*`Name`*`' followed by operator '`*`Name`*`' is confusing. Use parentheses.`** -- Some combinations of operators seem to be confusing. For example:

```
a = b – c – d;
a = b – c + d;
a = b / c / d;
a = b / c * d;
```

tend to befuddle the reader. To reduce confusion we recommend using parentheses to make the association of these operators explicit. For example:

```
a = (b – c) – d;
a = (b – c) + d;
a = (b / c) / d;
a = (b / c) * d;
```

in place of the above.

835 **`A zero has been given as [left/right] argument to operator '`*`Name`*`'`** -- A 0 has been provided as an operand to an arithmetic operator. The name of the operator is provided in the message as well as the side of the operator (**`left`** or **`right`**) that had the unusual value. For example:

```
n = n + 0 - m;
```

will produce a message that the right hand operand of **`operator '+'`** is zero.

In general the operators examined are the binary operators:

```
+ - * / % | & ^ << >>
```

and the unary operators **`-`** and **`+`**.

An enumeration constant whose value is 0 is permitted with operators:

```
+ - >> <<
```

Otherwise a message is issued. For example:

```
enum color { red,
             blue = red+100,        /* ok */
             green = red*0x10       /* 835 */
             };
```

The assignment operators that have an arithmetic or bitwise component, such as **|=**, are also examined. The message given is equivalent to that given with the same operator without the assignment component.

836 **Conceivable access of pointer pointing *Integer* bytes past nul character by operator '*String*'** -- A situation was detected where it appears remotely possible that a buffer is being accessed beyond the (nul-terminated) ***string*** that was placed in the buffer. An example of accessing beyond the nul character is shown in the example below:

```
char buf[20];
int k = 4;
strcpy( buf, "a" );
if( buf[k] ==  'a' ) ... // legal but suspect
```

In this particular case the access would be deemed 'likely' and a different but related message (**448**) would have been issued. This message (836) could be issued if there were some intervening code involving **k**. See also message **690**.

838 **Previously assigned value to variable '*Symbol*' has not been used** -- An assignment statement was encountered that apparently obliterated a previously assigned value that had never had the opportunity of being used. For example, consider the following code fragment:

```
y = 1;
if( n > 0 ) y = 2;
y = 4;              // Informational 838
...
```

Here we can report that the assignment of **4** to **y** obliterates previously assigned values that were not used. We, of course, cannot report anything unusual about the assignment of 2. This will assign over a prior value of **1** that so far had not been used but the existence of an alternative path means that the value of **1** can still be employed later in the code and is accepted for the time being as reasonable. It is only the final assignment that raises alarm bells. See also Warning message **438**.

839 **Storage class of symbol '*Symbol*' assumed static (*Location*)** -- A declaration for a symbol that was previously declared **static** in the same module was found without the '**static**' specifier. For example:

```
static void f();
```

```
extern void f();    // Info 839
void f() {}         // Info 839
```

By the rules of the language '**static**' wins and the symbol is assumed to have internal linkage. This could be the definition of a previously declared **static** function (as in line 3 of the above example) in which case, by adding the **static** specifier, you will inhibit this message. This could also be a redeclaration of either a function or a variable (as in line 2 of the above example) in which case the redeclaration is redundant.

840 **Use of nul character in a string literal** -- A nul character was found in a string literal. This is legal but suspicious and may have been accidental. This is because a nul character is automatically placed at the end of a string literal and because conventional usage and most of the standard library's string functions ignore information past the first nul character.

843 **Variable '*Symbol*' (*Location*) could be declared as const** -- A variable of **static** storage duration is initialized but never modified thereafter. Was this an oversight? If the intent of the programmer is to not modify the variable, it could and should be declared as **const** **[30, Item 3]**. See also message. **844**.

844 **Pointer variable '*Symbol*' (*Location*) could be declared as pointing to const** -- The data pointed to by a pointer of **static** storage duration is never changed (at least not through that pointer). It therefore would be better if the variable were typed pointer to **const** **[30, Item 3]**. See also message **843**.

845 **The [left/right] argument to operator '*Name*' is certain to be 0** -- An operand that can be deduced to always be **0** has been presented to an arithmetic operator in a context that arouses suspicion. The name of the operator is provided in the message as well as the side of the operator (**left** or **right**) that had the unusual value. For example:

```
n = 0;
k = m & n;
```

will produce a message that the right hand operand of operator '**&**' is certain to be zero.

The operands examined are the right hand sides of operators

```
+ - | ||
```

the left hand sides of operators

```
/ %
```

and both sides of operators

```
* & << >> &&
```

The reason that the left hand side of operator **+** (and friends) is not examined for zero is that zero is the identity operation for those operators and hence is often used as an initializing value. For example:

```
sum = 0;
for( ... )
    sum = sum + what_ever;       // OK, no message
```

The message is not issued for arithmetic constant zeros. Message **835** is issued in that instance.

The message is also suspended when the expression has side-effects. For example:

```
i = 0;
buf[i++] = 'A';
```

We don't consider it reasonable to force the programmer to write:

```
buf[0] = 'A';
i = 1;
```

846 **Signedness of bit-field is implementation defined** -- A bit-field was detected having the form:

```
int a:5;
```

Most bit fields are more useful when they are **unsigned**. If you want to have a **signed** bit field you must explicitly indicate this as follows:

```
signed int a:5;
```

The same also holds for typedef's. For example,

```
typedef int INT;
typedef signed int SINT;
struct  {
        INT a:16;   // Info 846
        SINT b:16;  // OK
        }:
```

It is very unusual in C or C++ to distinguish between **signed int** and just plain **int**. This is one of those rare cases.

847 **Thread *'Symbol'* has unprotected call to thread unsafe function *'Symbol'*** -- A thread named in the message makes an unprotected call (i.e., outside

of a critical section) on the function named in the message. The function had previously been identified as thread unsafe. See **Chapter 12 MULTI THREAD SUPPORT** for a definition of the terms: unprotected and thread unsafe.

This is not necessarily an error. Most thread unsafe functions may be called outside of critical sections provided no other thread is making such a call. There are other messages (at the Warning level) that will be issued when some other thread is also calling the same function, so it would normally be safe to suppress this message.

849 **Symbol *'Symbol'* has same enumerator value *'String'* as enumerator *'Symbol'*** -- Two enumerators have the same value. For example:

```
enum colors { red, blue, green = 1 };
```

will elicit this informational message. This is not necessarily an error and you may want to suppress this message for selected enumerators.

850 **for loop index variable *'Symbol'* whose type category is *'String'* modified in body of the for loop** -- The message is parameterized with a type categroy which is one of:

integral	some form of integer
float	some form of floating point number
string	some for of char * including wide char
pointer	some form of pointer other than string
enumeration	an enumeration of some kind
unclassified	none of the above

This will allow you to be more selective in delivery of messages because you may suppress or enable messages according to these classifications. For example:

```
-e850
+-string(850,integral)
+estring(850,float)
```

will enable Info 850 for integrals or for floats but not for other forms of loop variables.

864 **Expression involving variable '*Symbol*' possibly depends on order of evaluation** -- The variable cited in the message is either passed to a reference that is not a **const** reference or its address is passed to a pointer that is not a pointer to **const**. Hence the variable is potentially modified by the function. If the same variable is used elsewhere in the same expression, then the result may depend on the order of evaluation of the expression. For example:

```
int g( int );
int h( int & );
```

```
int f( int k )
    {
    return g(k) + h(k);   // Info 864
    }
```

Here the compiler is free to evaluate the call to **g()** first with the original value of **k** and then call **h()** where **k** gets modified. Alternatively, it can, with equal validity, call **h()** first in which case the value passed to **g()** would be the new value.

The object being modified could be the implicit argument (the **this** argument) to a member function call. For example:

```
void f( int, int );
class X { public: int bump(); int k; };
...
X x;
f( x.bump(), x.bump() );  // Info 864
```

Here the message states that the expression involving object **x** possibly depends on the order of evaluation. **x** is an implicit argument (by reference) to the **bump()** member function. If the member function **bump()** were declared **const,** then the message would not have been emitted. (See also 13.1 Order of Evaluation and Warning **564**)

865 ***user-determined text*** -- Message **865** is issued as a result of the **-message** option. For example:

```
#ifndef N
//lint -message(Please supply a definition for N)
#endif
```

will issue the message only if **N** is undefined. See option **-message**.

866 **Unusual use of '*String*' in argument to sizeof** -- An expression used as an argument to **sizeof()** counts as "unusual" if it is not a constant, a symbol, a function call, a member access, a subscript operation (with indices of zero or one), or a dereference of the result of a symbol, scoped symbol, array subscript operation, or function call. Also, since unary '**+**' could legitimately be used to determine the size of a promoted expression, it does not fall under the category of "unusual". Example:

```
char A[10];
unsigned end = sizeof(A - 1);          // 866: was 'sizeof(A) - 1' intended?
size_of_promoted_char = sizeof(+A[0]); // OK, + makes a difference
size_t s1 = sizeof( end+1 );           // 866: use +end to get promoted type
size_t s2 = sizeof( +(end+1) );        // OK, we won't complain
struct B *p;                           // B is some POD.
B b1;

memcpy( p, &b1, sizeof(&b1) );         // 866: sizeof(b1)intended?
```

```
size_t s3 = sizeof(A[0]);              // OK, get the size of an element.
size_t s4 = sizeof(A[2]);              // 866: Not incorrect, but unusual ...
```

19.6 C Elective Notes

Messages in the 900 level are termed "elective" because they are not normally on. They must be explicitly turned on with an option of the form **+e9**... or **-w4**. Messages in the range **910**-**919** involve implicit conversions. Messages in the range **920**-**930** involve explicit conversions (casts).

900 **Successful completion, '*Integer*' messages produced** -- This message exists to provide some way of ensuring that an output message is always produced, even if there are no other messages. This is required for some windowing systems. For this purpose use the option **+e900**

904 **Return statement before end of function '*Symbol*'** -- A return statement was found before the end of a function definition. Many programming standards require that functions contain a single exit point located at the end of the function. This can enhance readability and may make subsequent modification less error prone.

905 **Non-literal format specifier used (with arguments)** -- A **printf/scanf** style function received a non-literal format specifier but, unlike the case covered by Warning **592** the function also received additional arguments. E.g.

```
char *fmt;
int a, b;
...
printf( fmt, a, b );
```

Variable formats represent a very powerful feature of C/C++ but they need to be used judiciously. Unlike the case covered by Warning **592**, this case cannot be easily rewritten with an explicit visible format. But this Elective Note can be used to examine code with non-literal formats to make sure that no errors are present and that the formats themselves are properly constructed and contain no user-provided data. See Warning **592**.

909 **Implicit conversion from *Type* to bool** -- A non-bool was tested as a Boolean. For example, in the following function:

```
int f(int n)
    {
    if( n ) return n;
    else return 0;
    }
```

the programmer tests '**n**' directly rather than using an explicit Boolean expression such

as '**`n != 0`**'. Some shops prefer the explicit test.

910 **`Implicit conversion (`*`Context`*`) from 0 to pointer`** -- A pointer was assigned (or initialized) with a 0. Some programmers prefer other conventions such as **`NULL`** or **`nil`**. This message will help such programmers root out cavalier uses of 0. This is relatively easy in C since you can define NULL as follows:

```
#define NULL (void *)0
```

However, in C++, a **`void*`** cannot be assigned to other pointers without a cast. Instead, assuming that **`NULL`** is defined to be **`0`**, use the option:

```
--emacro((910),NULL)
```

This will inhibit message **`910`** in expressions using **`NULL`**. This method will also work in C.

Both methods assume that you expressly turn on this message with a **`+e910`** or equivalent.

911 **`Implicit expression promotion from `*`Type`*` to `*`Type`*** -- Notes whenever a sub-integer expression such as a **`char`**, **`short`**, **`enum`**, or bit-field is promoted to **`int`** for the purpose of participating in some arithmetic operation or function call.

912 **`Implicit binary conversion from `*`Type`*` to `*`Type`*** -- Notes whenever a binary operation (other than assignment) requires a type balancing. A smaller range type is promoted to a larger range type. For example: **`3 + 5.5`** will trigger such a message because **`int`** is converted to **`double`**.

913 **`Implicit adjustment of expected argument type from `*`Type`*` to `*`Type`*** -- Notes whenever an old-style function definition contains a sub-integer or **`float`** type. For example:

```
int f( ch, x ) char ch; float x; { ...
```

contains two **`913`** adjustments.

914 **`Implicit adjustment of function return value from `*`Type`*` to `*`Type`*** -- Notes whenever the function return value is implicitly adjusted. This message is given only for functions returning arrays.

915 **`Implicit conversion (`*`Context`*`) `*`Type`*` to `*`Type`*** -- Notes whenever an assignment, initialization or **`return`** implies an arithmetic conversion (***`Context`*** specifies which).

916 **`Implicit pointer assignment conversion (`*`Context`*`)`** -- Notes whenever an

assignment, initialization or **return** implies an implicit pointer conversion (***Context*** specifies which).

917 **Prototype coercion (*Context*) *Type* to *Type*** -- Notes whenever an implicit arithmetic conversion takes place as the result of a prototype. For example:

```
double sqrt(double);
... sqrt(3); ...
```

will elicit this message because **3** is quietly converted to **double**.

918 **Prototype coercion (*Context*) of pointers** -- Notes whenever a pointer is implicitly converted because of a prototype. Because of prototype conversion, **near** pointers will otherwise be silently mapped into **far** pointers. **far** pointers mapped into **near** pointers also generate message **619**.

919 **Implicit conversion (*Context*) *Type* to *Type*** -- A lower precision quantity was assigned to a higher precision variable as when an **int** is assigned to a **double**.

920 **Cast from *Type* to void** -- A cast is being made from the given type to **void**.

921 **Cast from *Type* to *Type*** -- A cast is being made from one integral type to another.

922 **Cast from *Type* to *Type*** -- A cast is being made to or from one of the floating types (**float**, **double**, **long double**).

923 **Cast from *Type* to *Type*** -- A cast is being made either from a pointer to a non-pointer or from a non-pointer to a pointer.

924 **Cast from *Type* to *Type*** -- A cast is being made from a **struct** or a **union**. If the cast is not to a compatible **struct** or **union** error **69** is issued.

925 **Cast from pointer to pointer** -- A cast is being made to convert one pointer to another such that one of the pointers is a pointer to **void**. Such conversions are considered harmless and normally do not even need a cast.

926 **Cast from pointer to pointer** -- A cast is being made to convert a **char** pointer to a **char** pointer (one or both of the **char**'s may be **unsigned**). This is considered a 'safe' cast.

927 **Cast from pointer to pointer** -- A cast is being made to convert a **char** (or **unsigned char**) pointer to a non-**char** pointer. **char** pointers are sometimes implemented differently from other pointers and there could be an information loss in such a conversion.

928 **`Cast from pointer to pointer`** -- A cast is being made from a non-**`char`** pointer to a **`char`** pointer. This is generally considered to be a 'safe' conversion.

929 **`Cast from pointer to pointer`** -- A cast is being made to convert one pointer to another that does not fall into one of the classifications described in **925** through **928** above. This could be nonportable on machines that distinguish between pointer to **`char`** and pointer to word. Consider casting a pointer to pointer to **`char`** to a pointer to pointer to word. The indirect bit pattern remains unchanged.

930 **`Cast from`** ***`Type`*** **`to`** ***`Type`*** -- A cast is being made to or from an enumeration type.

931 **`Both sides have side effects`** -- Indicates when both sides of an expression have side-effects. An example is **`n++ + f()`**. This is normally benign. The really troublesome cases such as **`n++ + n`** are caught via Warning **564**.

932 **`Passing near pointer to library function '`*`Symbol`*`' (`*`Context`*`)`** -- A source of error in Windows programming is to pass a **`near`** pointer to a library function (See Chapter 6. LIBRARIES). If the library is a DLL library, then in supplying the missing segment, the library would assume its own data segment, which would probably be wrong. See also messages **933** and **934**.

933 **`Passing near pointer to far function (`*`Context`*`)`** -- A source of error in Windows programming is to pass a **`near`** pointer to a DLL function. Most Microsoft functions in DLLs are declared with the **`far`** modifier. Hence this can be tentatively used as a discriminant to decide that a pointer is too short. An advantage that this Note has over **932** is that it can catch functions designated only by pointer. Also you may be using libraries that are not DLLs and that share the same DS segment. In this case, **932** may produce too many superfluous messages. See also message **934**.

934 **`taking address of near auto variable '`*`Symbol`*`' (`*`Context`*`)`** -- A source of error in writing DLL libraries is that the stack segment may be different from the data segment. In taking the address of a **`near`** data object only the offset is obtained. In supplying the missing segment, the compiler would assume the data segment, which could be wrong. See also messages **932** and **933**.

935 **`int within struct`** -- This Note helps to locate non-portable data items within **`struct`**'s. If instead of containing **`int`**'s and **`unsigned int`**'s, a **`struct`** were to contain **`short`**'s and **`long`**'s then the data would be more portable across machines and memory models. Note that bit fields and **`union`**'s do not get complaints.

936 **`old-style function definition for function '`*`Symbol`*`'`** -- An "old-style" function definition is one in which the types are not included between parentheses. Only names are provided between parentheses with the type information following the right parenthesis. This is the only style allowed by K&R.

937 **`old-style function declaration for function 'Symbol'`** -- An "old-style" function declaration is one without type information for its arguments.

938 **`parameter 'Symbol' not explicitly declared`** -- In an "old-style" function definition it is possible to let a function parameter default to **`int`** by simply not providing a separate declaration for it.

939 **`return type defaults to int for function 'Symbol'`** -- A function was declared without an explicit return type. If no explicit storage class is given, then Informational **`745`** is also given provided the Deduce Return mode flag (**`fdr`**) is off. This is meant to catch all cases.

940 **`omitted braces within an initializer`** -- An initializer for a subaggregate does not have braces. For example:

```
int a[2][2] = { 1, 2, 3, 4 };
```

This is legal C but may violate local programming standards. The worst violations are covered by Warning **`651`**.

941 **`Result 0 due to operand(s) equaling 0 in operation 'String'`** -- The result of a constant evaluation is **`0`** owing to one of the operands of a binary operation being **`0`**. This is less severe than Info **`778`** wherein neither operand is **`0`**. For example, expression **`(2&1)`** yields a **`778`** whereas expression **`(2&0)`** yields a **`941`**.

942 **`Possibly truncated addition promoted to float`** -- An integral expression (signed or unsigned) involving addition or subtraction is converted to a floating point number. If an overflow occurred, information would be lost. See also messages **`647`**, **`776`** and **`790`**.

943 **`Too few initializers for aggregate`** -- The initializer **`{0}`** was used to initialize an aggregate of more than one item. Since this is a very common thing to do, it is given a separate message number, which is normally suppressed. See **`785`** for more flagrant abuses.

944 **`[left/right] argument for operator 'String' always evaluates to [True/False]`** -- The indicated operator (given by ***`String`*** has an argument that appears to always evaluate to either '**`True`**' or '**`False`**' (as indicated in the message). This is given for Boolean operators (**`||`** and **`&&`** and for Unary operator **`!`**) and information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message **`506`**, which is based on testing constants or combinations of constants.

945 **`Undefined struct used with extern`** -- Some compilers refuse to process

declarations of the form:

```
extern struct X s;
```

where struct **X** is not yet defined. This note can alert a programmer porting to such platforms.

946 **Relational or subtract operator applied to pointers** -- A relational operator (one of **>**, **>=**, **<**, **<=**) or the subtract operator has been applied to a pair of pointers. The reason this is of note is that when large model pointers are compared (in one of the four ways above) or subtracted, only the offset portion of the pointers is subject to the arithmetic. It is presumed that the segment portion is the same. If this presumption is not accurate then disaster looms. By enabling this message you can focus in on the potential trouble spots.

947 **Subtract operator applied to pointers** -- An expression of the form **p - q** was found where both **p** and **q** are pointers. This is of special importance in cases where the maximum pointer can overflow the type that holds pointer differences. For example, suppose that the maximum pointer is 3 Gigabytes -1, and that pointer differences are represented by a long, where the maximum long is 2 Gigabytes -1. Note that both of these quantities fit within a 32 bit word. Then subtracting a small pointer from a very large pointer will produce an apparent negative value in the long representing the pointer difference. Conversely, subtracting a very large pointer from a small pointer can produce a positive quantity.

The alert reader will note that a potential problem exists whenever the size of the type of a pointer difference equals the size of a pointer. But the problem doesn't usually manifest itself since the highest pointer values are usually less than what a pointer could theoretically hold. For this reason, the message cannot be given automatically based on scalar types and hence has been made an Elective Note.

Compare this Note with that of **946**, which was designed for a slightly different pointer difference problem.

948 **Operator '*String*' always evaluates to [True/False]**-- The operator named in the message is one of four relational operators or two equality operators in the list:

```
>    >=    <    <=
==   !=
```

The arguments are such that it appears that the operator always evaluates to either **True** or to **False** (as indicated in the message). This is similar to message **944**. Indeed there is some overlap with that message. Message **944** is issued in the context where a Boolean is expected (such as the left hand side of a **?** operator) but may not involve a relational operator. Message **948** is issued in the case of a relational (or equality)

operator but not necessarily in a situation that requires a Boolean.

950 **Non-ISO/ANSI reserved word or construct: '*Symbol*'** -- ***Symbol*** is either a reserved word that is non-ISO/ANSI or a construct (such as the **//** form of comment in a C module). This Elective Note is enabled automatically by the **-A** option. If these messages are occurring in a compiler or library header file over which you have no control, you may want to use the option **-elib(950)**. If the reserved word is one you want to completely disable, then use the option **-rw(*Word*)**.

951 **Pointer to incomplete type '*Symbol*' employed in operation** -- A pointer to an incomplete type (for example, struct **X** where struct **X** has not yet been defined in the current module) was employed in an assignment or in a comparison (for equality) operator. For example, suppose a module consisted only of the following function:

```
struct A * f(struct A *p )
    {
    return p;
    }
```

Since **struct A** had not been defined, this message will be issued. Such employment is permitted by the standard but is not permitted by all C compilers. If you want to deploy your application to the maximum number of platforms, you should enable this Elective Note.

952 **Parameter '*Symbol*' (*Location*) could be declared const** -- A parameter is not modified by a function. For example:

```
int f( char *p, int n ) { return *p = n; }
```

can be redeclared as:

```
int f( char * const p, const int n ) { return *p = n; }
```

There are few advantages to declaring an unchanging parameter a **const**. It signals to the person reading the code that a parameter is unchanging, but, in the estimate of most, reduces legibility. For this reason the message has been given an Elective Note status.

However, there is a style of programming that encourages declaring parameters **const**. For the above example, this style would declare **f** as

```
int f( char *p, int n);
```

and would use the **const** qualifier only in the definition. Note that the two forms are compatible according to the standard. The declaration is considered the interface specification where **const** does not matter. The **const** does matter in the definition of

the function, which is considered the implementation. Message **952** could be used to support this style.

Marking a parameter as **const** does not affect the type of argument that can be passed to the parameter. In particular, it does not mean that only **const** arguments may be passed. This is in contrast to declaring a parameter as pointer to **const** or reference to **const**. For these situations, Informational messages are issued (**818** and **1764** respectively) and these do affect the kinds of arguments that may be passed. See also messages **953** and **954**.

953 **Variable '*Symbol*' (*Location*) could be declared as const** -- A local auto variable was initialized and referenced but never modified. Such a variable could be declared **const**. One advantage in making such a declaration is that it can furnish a clue to the program reader that the variable is unchanging. Other situations in which a **const** can be added to a declaration are covered in messages **818**, **834**, **844**, **952**, **954** and **1764**.

954 **Pointer variable '*Symbol*' (*Location*) could be declared as pointing to a const** -- The data pointed to by a pointer is never changed (at least not through that pointer). It may therefore be better, or at least more descriptive, if the variable were typed pointer to **const**. For example:

```
{
char *p = "abc";
for( ; *p; p++ ) print(*p);
}
```

can be redeclared as:

```
{
const char *p = "abc";
for( ; *p; p++ ) print(*p);
}
```

It is interesting to contrast this situation with that of pointer parameters. The latter is given Informational status (**818**) because it has an effect of enhancing the set of pointers that can be passed into a function. Other situations in which a **const** can be added to a declaration are covered in messages **952**, **953** and **1764**.

955 **Parameter name missing from prototype for function '*Symbol*'** -- In a function declaration a parameter name is missing. For example:

```
void f(int);
```

will raise this message. This is perfectly legal but misses an opportunity to instruct the user of a library routine on the nature of the parameter. For example:

```
void f(int count);
```

would presumably be more meaningful. **[27, Rule 34]**

This message is not given for function definitions, only function declarations.

956 **Non const, non volatile static or external variable '*Symbol*'** -- This check has been advocated by programmers whose applications are multi-threaded. Software that contains modifiable data of static duration is often non-reentrant. That is, two or more threads cannot run the code concurrently. By 'static duration' we mean variables declared static or variables declared external to any function. For example:

```
int count = 0;
void bump() { count++; }
void get_count() { return count; }
```

If the purpose is to obtain a count of all the **bump()**'s by a given thread then this program clearly will not do since the global variable **count** sums up the **bump()**'s from all the threads. Moreover, if the purpose of the code is to obtain a count of all **bump()**'s by all threads, it still may contain a subtle error (depending on the compiler and the machine). If it is possible to interrupt a thread between the access of **count** and the subsequent store, then two threads that are **bump()**'ing at the same time, may register an increase in the **count** by just one.

Please note that not all code is intended to be re-entrant. In fact most programs are not designed that way and so this Elective Note need not be enabled for the majority of programs. If the program is intended to be re-entrant, all uses of non-const static variables should be examined carefully for non-reentrant properties.

957 **Function '*Symbol*' defined without a prototype in scope** -- A function was defined without a prototype in scope. It is usually good practice to declare prototypes for all functions in header files and have those header files checked against the definitions of the function to assure that they match.

If you are linting all the files of your project together such cross checking will be done in the natural course of things. For this reason this message has been given a relatively low urgency of Elective Note.

958 **Padding of *Integer* byte(s) is required to align *string* on *Integer* byte boundary** -- This message is given whenever padding is necessary within a **struct** to achieve a required member alignment. ***String*** designates that which is being aligned. Consider:

```
struct A { char c; int n; };
```

Assuming that **int** must be aligned on a 4-byte boundary and assuming the size of a **char** to be 1, then this message will be issued indicating that there will be a padding of 3 bytes to align the number.

The alignment requirements vary with the compiler, the machine and, sometimes, compiler options. When separately compiled programs need to share data at the binary level it helps to remove any artificially created padding from any of the structures that may be shared.

959 **struct size (*Integer* bytes) is not an even multiple of the maximum member alignment (*Integer* bytes)** -- The alignment of a structure (or union) is equal to the maximum alignment of any of its members. When an array of structures is allocated, the compiler ensures that each structure is allocated at an address with the proper alignment. This will require padding if the size of the structure is not an even multiple of its maximum alignment. For example:

```
struct A { int n; char ch; } a[10];
```

Assuming the size and alignment of **int** is 4 then the size of each struct is 5 but its alignment is 4. As a result each struct in the array will be padded with 3 bytes.

Alignment can vary with the compiler and the machine. If binary data is to be shared by separately compiled modules, it is safer to make sure that all shared structures and unions are explicitly padded.

960 **Violates MISRA *Year* Required Rule *Name*, *String*** -- MISRA is the "Guidelines for the use of the C Language in Vehicle Based Software". **[9, 33]** The first version of the MISRA Standard was released in 1998 and the second in 2004. Lint refers to the rules below in the form:

Rule ***I*/*D***

where ***I*** is an integer rule number from the 1998 Standard and ***D*** is a decimal rule number from the 2004 Standard. When issuing messages, Lint will refer to the rules using the form:

Rule ***I***

for the 1998 Standard and the form:

Rule ***D***

for the 2004 Standard.

The list of required checks made for both MISRA 1998 and 2004 are:

(Rule 19/7.1) Octal constant used.
(Rule 32/9.3) Should initialize either all enum members or only the first.
(Rule 33/12.4) Side effects on right hand side of logical operator.
(Rule 42/12.10) Comma operator used outside of '**for**' expression.
(Rule 54/14.3) Null statement not in line by itself.
(Rule 57/14.5) **continue** statement should not be used.
(Rules 59/14.8 & 14.9) Left brace expected for **if**, **else**, **for**, **do**, and **while**.
(Rule 65/13.4) Floating point variable used as loop counter.
(Rule 68/8.6) Function not declared at file scope.
(Rule 69/16.1) Function has variable number of arguments.
(Rule 73/16.3) Either all parameters or no parameters should have identifiers.
(Rule 91/19.5 '**#define/#undef**' used within a block.
(Rule 98/19.12) Multiple use of '#' and/or '##' operators in macro definition.
(Rule 100/19.14) Non-standard use of '**defined**' preprocessor operator.

Required checks made exclusively for MISRA 1998 are:

(Rule 58) **break** used outside of a switch.
(Rule 88) Header file name contains non-standard character.
(Rule 110) Bitfields inside union.

Required checks made exclusively for MISRA 2004 are:

(Rule 8.5) No definitions of objects or function in header files.
(Rules 10.1 & 10.2) Prohibited implicit conversion.
(Rules 10.3 & 10.4) Prohibited cast of complex expressions.
(Rule 10.5) Recasting required for operators '**~**' and '**<<**'.
(Rule 12.3) '**sizeof**' used on expressions with side effect.
(Rule 12.7) Bitwise operator applied to signed underlying type.
(Rule 12.9) Prohibited operator applied to unsigned underlying type.
(Rule 14.6) More than one '**break**' terminates loop.
(Rule 14.10) No '**else**' at end of '**if ... else if**' chain.
(Rule 15.4) Boolean value in switch expression.
(Rule 18.4) Unions shall not be used.
(Rule 19.6) Use of '**#undef**' prohibited.

MISRA 1998 checking is achieved using the **-misra(1)** option. For MISRA 2004 checks, use **-misra(2)**.

For messages **960**, **961**, **1960** and **1963**, you may disable individual rules to your taste by using the message number and Rule number in an **esym** option. For example:

```
-esym( 960, 75, 8? )
```

will suppress MISRA rules 75 and any of those between 80 and 89 inclusive that are issued as the result of a 960. See **[33, 34]** for information on the MISRA guidelines.

See **Section 13.12 MISRA Standards Checking** for information on activating Lint's MISRA checking in general.

961 **Violates MISRA *Year* Advisory Rule *Name*, *String*** -- This message is issued for some violations of the MISRA advisory guidelines. Certain rules were advisories in the 1998 Standard and became required for the 2004 Standard and vice versa. Therefore, you might see some rules here that are already listed above for message 960.

The list of advisory checks made for both MISRA 1998 and 2004 are:

(Rule 47/12.1) Dependence placed on C's operator precedence.
(Rule 87.19.1) Only preprocessor statements and comments before '**#include**'.
(Rule 93/19.7) Use of function-like macros is discouraged.
(Rule102/17.5) More than two pointer indirection levels used.

Advisory checks made exclusively for MISRA 1998 are:

(Rule 18) Constant requires numerical suffix.
(Rule 28) '**register**' class discouraged.
(Rule 40) '**sizeof**' used on expressions with side effect.
(Rule 44) Redundant explicit casting.
(Rule 55) Non-case label.
(Rule 60) No '**else**' at end of '**if ... else if**' chain.
(Rule 63) Boolean value in switch expression.
(Rule 92) Use of '**#undef**' is discouraged.

Advisory checks made exclusively for MISRA 2004 are:

(Rule 19.2) Header file name contains non-standard character.
(Rule 19.13) No use of '#' or '##'.

This message can be suppressed based on rule number. See also Message **960**.

See **[33, 34]** for information on the MISRA guidelines. See **Section 13.12 MISRA Standards Checking** for information on activating Lint's MISRA checking in general.

962 **Macro '*Symbol*' defined identically at another location (*Location*)** -- The same macro was defined in the same way in two different places in the source code. This is not a good practice since a subsequent change to one of the macros could lead to confusion.

963 **Qualifier const or volatile follows/precedes a type; use -fqb/+fqb to reverse the test** -- The declarations in the following example are equivalent:

```
//lint +e963  report on qualifier-type inversion
extern const char *p;
extern char const *p;   // Note 963
```

The qualifier '**const**' and '**volatile**' may appear either before or after or even between other declaration specifiers. Many programmers prefer a consistent scheme such as always placing the qualifier before the type. If you enable 963 (using **+e963**) this is what you will get by default. The message will contain the word '**follows**' rather than the word '**precedes**'.

There is a diametrically opposite convention, viz. that of placing the qualifier after the type. As the message itself reminds the user you will obtain the reverse test if you turn off the **fqb** (place qualifiers before types) flag. Thus

```
//lint -fqb   turn off the Qualifiers Before types flag
//lint +e963  report on type-qualifier inversion
extern const char *p;   // Note 963
extern char const *p;
```

Note that the use of this flag will cause '**follows**' in the message to be replaced by '**precedes**' and the alternative option mentioned within the '**use**' clause is changed to its opposite orientation.

Dan Saks **[36]** and Vandevoorde and Josuttis, **[32, section 1.4]**, (**Chapter 21. Bibliography**) provide convincing evidence that this alternative convention is indeed the better one.

964 **Header file *'FileName'* not directly used in module *'String'*** -- The given header file was not used in the given module, however it, itself, included a header file (possibly indirectly) that was used. An example of this is **os2.h** that is an umbrella header serving only to include other headers. Compare this message with **766**. See also the discussion in Section 13.8.1 Unused Headers.

966 **Indirectly included header file '*FileName*' not used in module '*String*'** -- The header file given by ***FileName*** was unused directly or indirectly in a given module outside of its group. It was not, however, directly included by the module and so may not easily be excluded without disturbing the header including it. Since this header may be included in other places, caution is advised. This message is a weaker version of **766**. See the discussion in Section 13.8.1 Unused Headers. See also message **964**.

967 **Header file '*FileName*' does not have a standard include guard** -- You may protect against the repeated inclusion of headers by means of a standard include guard having the following form:

```
#ifndef Name
```

```
#define Name
...
#endif
```

The header file cited in the message does not have such a guard. It is standard practice in many organizations to always place include guards within every header.

See message **451** for more information about header include guards.

970 **Use of modifier or type '*Name*' outside of a typedef** -- Some standards require the use of type names (defined in **typedef**'s) in preference to raw names used within the text of the program. For example they may want you to use **INT32** rather than **int** where **INT32** is defined as:

```
typedef int INT32;
```

This message is normally issued for the standard intrinsic types: **bool**, **char**, **wchar_t**, **int**, **float**, **double**, and for modifiers **unsigned**, **signed**, **short** and **long**. You may enable this message and then suppress the message for individual types to obtain special effects. For example the following will enable the message for all but **bool**.

```
+e970  -esym(970,bool)
```

971 **Use of 'char' without 'signed' or 'unsigned'** -- The '**char**' type was specified without an explicit modifier to indicate whether the **char** was **signed** or **unsigned**. The plain **char** type can be regarded by the compiler as identifying a **signed** or an **unsigned** quantity, whichever is more efficient to implement. Because of this ambiguity, some standards do not like the use of **char** without an explicit modifier to indicate its signedness.

973 **Unary operator in macro '*Symbol*' not parenthesized** -- A unary operator appearing in an expression-like macro was found to be not parenthesized. For example:

```
#define N -1
```

The user may prefer to parenthesize such things as:

```
#define N (-1)
```

This has been placed in the elective note category because we cannot find an instance when this really produces a problem. The important case of unparenthesized binary operators is covered with message **773**.

974 **Worst case function for stack usage: *String*** -- This message, issued at global wrap-up, will report on the function that requires the most stack. The stack

required consists of the amount of auto storage the function requires plus the amounts required in any chain of functions called. The worst case chain is always reported.

To obtain a report of all the functions, use the **+stack** option.

Reasonable allowances are made for function call overhead and the stack requirements of external functions. These assumptions can be controlled via the **+stack** option.

If recursion is detected it will be reported here, as this is considered worse than any finite case. The next worse case is that the stack can't be determined because a function makes a call through a function pointer. The function is said to be non-deterministic. If neither of these conditions prevail, the function that heads the worst case chain of calls will be reported upon.

The message will normally provide you with the name of a called function. If the function is recursive this will provide you with the first call of a recursive loop. To determine the full loop, you will need a full stack report as obtained with the **+stack** option. You need a suboption of the form **&file=*file*** to specify a file which will contain a record for each function for which a definition was found. You will be able to follow the chain of calls to determine the recursive path.

If you can assure yourself through code analysis that there is an upper bound to the amount of stack utilized by some recursive function, then you can employ the **+stack** option to specify the bound for this function. The function will no longer be considered recursive but rather finite. In this way, possibly through a sequence of options, you can progressively eliminate apparent recursion and in that way arrive at a safe upper bound for stack usage. Similar considerations apply for non-deterministic functions.

975 **Unrecognized pragma '*Name*' will be ignored** -- The first identifier after **#pragma** is considered the name of the pragma. If the name is unrecognized then the remainder of the line is ignored. Since the purpose of **#pragma** is to allow for compiler-dependent communication, it is not really expected that all pragmas will be understood by all third-party processors of the code. Thus, this message does not necessarily indicate that there is anything wrong and could easily be suppressed entirely.

Moreover, if the pragma occurs in a library header, this message would not normally be issued because the option **-wlib(1)** would be in effect (this option is present in all of our compiler options files).

But, if the pragma occurs in user code, then it should be examined to see if there is something there that might interest a lint processor. There are a variety of facilities to deal with pragmas; in particular, they can be mapped into linguistic constructs or lint options or both. See **Section 5.8.7.2 User Pragmas**.

19.7 C++ Syntax Errors

1001 **Scope '*Name*' must be a struct or class name** -- In an expression of the form **X::Y**, **X** must be a class name. **[11 section 10.4]**

1002 **'this' must be used in class member function** -- The keyword **this** refers to the class being passed implicitly to a member function. It is invalid outside a class member function. **[11 section 5.1]**

1003 **'this' may not be used in a static member function** -- A **static** member function receives no **this** pointer. **[11 section 9.5]**

1004 **Expected a pointer to member after .* or ->*** -- The **.*** and **->*** operators require pointer to members on the right hand side. **[11 section 5.5]**

1005 **Destructor declaration requires class** -- While expecting a declaration a '**~**' character was encountered. This was presumed to be the start of a destructor. However no class was specified. **[11 section 12.4]**

1006 **Language feature '*String*' not supported** -- The indicated feature, while not supported in the current version, will hopefully be supported in future versions of the product.

1007 **Pure specifier for function '*Symbol*' requires a virtual function** -- An '**=**' was found after a declaration. Was this the start of a pure specifier? The declaration was not that of a member function, which it must be. Also, the member function should be virtual. **[11 section 10.3]**

1008 **Expected '0' to follow '=', text ignored** -- Some nonstandard extensions to C++ allow integers to follow '**=**' for declarations of member functions. If you are using such extensions simply suppress this message. If only library headers are using this extension use **-elib(1008)**. **[11 section 10.3]**

1009 **operator '*String*' not redefinable** -- The cited operator, one of '

```
.* ? .
```

may not be overloaded. **[11 section 13.4]**

1010 **Expected a type or an operator** -- Following the keyword **operator** the parser expected either an operator (including **new**, **delete**, **()**, **[]**, comma) or a type. **[11 sections 13.4 and 12.3.2]**

1011 **Conversion Type Name too long** -- An upper limit of 50 characters has been

reached on a conversion type name.

1012 **`Type not needed before 'operator type'`** -- The return type of a function introduced with '**`operator`** ***`Type`***' is ***`Type`*** and may not be preceded with the same or any other type. **[11 section 12.3.2]**

1013 **`Symbol '`*`Name`*`' not a member of class '`*`Name`*`'`** -- The second operand of a scope operator or a '**`.`**' or '**`->`**' operator is not a member of the **`class`** (**`struct`** or **`union`**) expressed or implied by the left hand operand. **[11 section 3.2]**

1014 **`Explicit storage class not needed for member function '`*`Symbol`*`'`** -- An explicit ***`Symbol`*** storage class such as **`extern`** or **`static`** was given in a separate definition of a class member. The storage class is effectively defined by its appearance within the class and may not be restated at definition time.

1015 **`Symbol '`*`Name`*`' not found in class`** -- In an expression of the form **`X::Y`**, **`Y`** must be a member of **`X`** or of a public or protected base class of **`X`**. **[11 section 10.4]**

1016 **`Symbol '`*`Symbol`*`' is supposed to denote a class`** -- In a ***`base-specifier`*** an identifier is supposed to specify a base class. However, the identifier was not previously declared in this module. **[11 section 10]**

1017 **`conflicting access-specifier '`*`String`*`'`** -- Two different access specifiers were given in a simple ***`base-specifier`***. **[11 section 10]**

1018 **`Expected a type after 'new'`** -- In an expression involving **`new`**, a type is expected after possibly processing a ***`placement`***. None was found. **[11 section 5.3.3]**

1019 **`Could not find match for function '`*`Symbol`*`(`*`String`*`)'`** -- In attempting to find a match between a set of overloaded functions or operators (name given as ***`Symbol`***) and an actual argument list (provided as ***`String`***) no match could be found. **[11 section 13.2]**

1020 **`template specialization for '`*`Symbol`*`' declared without a 'template<>' prefix`** -- A class template specialization is generally preceded by a '**`template`**' clause as in:

```
template< class T > class A { };       // a template
template<> class A<int> { };           // a specialization
```

If the '**`template`**' is omitted you will get this message but it will still be interpreted as a specialization. Before the standardization of template syntax was completed, a template specialization did not require this clause and its absence is still permitted by some compilers.

1022 **`Function: 'String' must be a class member`** -- There are four operators not to be defined except as class members. These are:

```
= () [] ->
```

The parameter ***`String`*** indicates which it is. **[11 sections 13.4.3 and 13.4.6]**

1023 **`Call ‘Name’ is ambiguous; candidates: ‘String’`** -- A call to an overloaded function or operator is ambiguous. The candidates of choice are provided in the message. **[11 section 13.2]**

1024 **`No function has same argument count as 'Invocation'`** -- A call to an overloaded function could not be resolved successfully because no function is declared with the same number of arguments as in the call. **[11 section 13.2]**

1025 **`No function matches invocation 'Name' on arg no. Integer`** -- A call to an overloaded function could not be resolved because each declared function has a type incompatibility with the indicated argument. **[11 section 13.2]**

1026 **`Undominated function 'String' does not dominate 'String' on call to 'String'`** -- A call to an overloaded function could not be resolved because no one function dominates all others. This is a subtle issue in the overload resolution process. The selected function must be strictly better than any non-selected function in at least one argument. **[11 section 13.2]**

1027 **`Non-consecutive default arguments in function 'String', assumed 0`** -- Default arguments need to be consecutive. For example

```
void f(int i=0, int j, int k=0);
```

is illegal. **[11 section 8.2.6]**

1028 **`Last argument not default in first instance of function 'String', assumed 0`** -- If any argument of a function is given a default value then all subsequent arguments need to be given a default value. **[11 section 8.2.6]**

1029 **`Default argument repeated in function 'String'`** -- A default value for a given argument for a given function should be given only once. **[11 section 8.2.6]**

1030 **`Not all arguments after arg no. Integer are default in function 'String'`** -- An argument that has a default value must either be followed by another argument that has a default value, or must be the last argument. **[11 section 8.2.6]**

1031 **`Local variable 'Symbol' used in default argument expression`** -- Default values for arguments may not use local variables. **[11 section 8.2.6]**

1032 **Member '*String*' cannot be called without object** -- There was an attempt to call a non-static member function without specifying or implying an object that could serve as the basis for the **this** pointer. If the member name is known at compile time it will be printed with the message. **[11 section 5.24]**

1033 **Static member functions cannot be virtual** -- You may not declare a **static** member function **virtual**. **[11 section 10.2]**

1034 **Static member '*Symbol*' is global and cannot be redefined** -- This can come as a surprise to the novice C++ programmer. The word '**static**' within a class definition is used to describe a member that is alone and apart from any one object of a class. But such a member has program scope not file scope. The word '**static**' outside a class definition implies file scope not program scope. **[11 section 9.5]**

1035 **Non-static member '*Symbol*' cannot initialize a default argument** -- A default argument cannot be initialized from a class member unless an instantiation of the class is provided. **[11 section 8.2.6]**

1036 **ambiguous reference to constructor; candidates: '*String*'** -- There is more than one constructor that can be used to make a desired conversion. **[11 section 12.3.2]**

1037 **ambiguous reference to conversion function; candidates: '*String*'** -- There is more than one conversion function (of the form **operator** ***type*** **()**) that will perform a desired conversion. **[11 section 12.3.2]**

1038 **type *Name* not found, nested type '*Name::String*' assumed** -- We have found what appears to be a reference to a type but no such type is in scope. We have, however, been able to locate a type buried within another class. Is this what the user intended? If this is what is intended, use full scoping. If your compiler doesn't support the scoping, suppress with -**esym**. **[11 section 3.2]**

1039 **Symbol '*Symbol*' is not a member of class '*String*'** -- In a declaration for the symbol **X::Y**, **Y** was not previously established as a member of **X**. **[11 section 10.4]**

1040 **Symbol '*Symbol*' is not a legal declaration within class '*String*'** -- A declaration of the symbol **X::Y** appears within a class definition (other than for class **X**). It is not a **friend** declaration. Therefore it is in error.

1041 **Can't declare '*String*', assumed 'operator *String*'** -- This message can be given with ***String*** equal to **new** or **delete**. Each '***String***' in this message has the same value. A common mistake with beginning C++ programmers is to declare (and/or define) **new** when they mean to define **operator new**. We presume this was what was

intended. **[11 section 12.5]**

1042 **`At least one class-like operand is required with Name`** -- In defining (or declaring) an operator you must have at least one class as an operand. **[11 section 13.4]**

1043 **`Attempting to 'delete' a non-pointer`** -- An expression being **`delete`**'d is a non-pointer, non-array. You may only **`delete`** what was created with an invocation of **`new`**. **[11 section 5.3.4]**

1046 **`member 'Symbol', referenced in a static function, requires an object`** -- The ***`Symbol`*** is a non-static member of a class and hence requires a class instantiation. None is in sight. **[10 section class.static]**

1047 **`a template declaration must be made at file scope`** -- A **`template`** declaration may not appear within a function or within a class. **[10 section temp.param]**

1048 **`expected a constant expression`** -- Within a template argument list a constant expression was expected. An expression of the form ***`T<arg1,arg2,...>`*** was encountered and ***`argi`*** for some ***`i`*** corresponds to a non-**`class`** parameter in the original template declaration. Such arguments need to be constants. **[10 section temp.decls]**

1049 **`Too many template arguments`** -- There are more arguments in the template class-name than there were parameters in the original template declaration. **[10 section temp.decls]**

1050 **`expected a template argument list '<...>' for template 'Symbol'`** -- The name of a class template identified by ***`Symbol`*** was used without specifying a template argument list. **[10 section temp.decl]**

1051 **`Symbol 'Name' is both a function and a variable`** -- Whereas it is possible to overload a function name by giving it two different parameter lists, it is not possible to overload a name in any other way. In particular, a function name may not also be used as a variable name. **[11 section 9.2]**

1052 **`a type was expected, 'class' assumed`** -- A template parameter list may consist of two kinds of parameters: type-parameters and parameter declarations. Type-parameters begin with one of the keywords **`class`**, **`typename`**, or **`template`**. Parameter declarations begin with a type. None of these conditions was detected. **[10 section temp.decl]**

1053 **`'String' cannot be distinguished from 'String'`** -- An overloaded function name had two parameter lists that were so close that discrimination between them would be difficult and error prone. Eg. **`void f(const int);`** and **`void`**

`f(int);` [11 section 13]

1054 **`template variable declaration expects a type, int assumed`** -- An expression of the form ***`T<arg,arg,...>`*** was encountered. One of the arguments corresponding to a type parameter in the original template declaration is not a type. **[10 section temp.class]**

1055 **`Symbol '`*`Symbol`*`' undeclared, assumed to return int`** -- Whereas in C you may call a function without a prior declaration, in C++ you must supply such a declaration. For C programs you would have received an Informational message (**718**) in this event. **[11 section 5.2.2]**

1056 **`assignment from void * is not allowed in C++`** -- Whereas in C you may assign from **`void*`** to any other (data) pointer without a diagnostic, in C++ you may not do this. It will require a cast. **[11 section 4.6]**

1057 **`member '`*`Symbol`*`' cannot be used without an object`** -- The indicated member referenced via scope operator cannot be used in the absence of a **`this`** pointer. **[11 section 5.2.4]**

1058 **`Initializing a non-const reference '`*`Symbol`*`' with a non-lvalue`** -- A reference is normally initialized with an lvalue. If you attempt to initialize a reference with a non-lvalue, a temporary is created to serve as a surrogate lvalue. However, modifications made to the temporary will be lost. This was legal at one time and is now illegal. Make the reference a **`const`** if you can. You may be initializing a reference without realizing it. A member function has an implicit parameter, which is taken to be a reference to its object. If this is the situation make the member **`const`**. That is, use **`void f(...) const;`** rather than **`void f(...);`**

1059 **`Can't convert from `*`Type`*` to `*`Type`** -- An attempt was made to initialize a reference with an object having a type other than the target type but no function could be found to effect the required conversion. **[11 section 12.3]**

1060 ***`String`*** **`member `*`Symbol`*` is not accessible to non-member non-friend functions`** -- There is an attempt to access a **`private`** or **`protected`** member of a class and the access is considered a violation of the access rules (although everything else proceeds as though no violation occurred). Specifically, the function attempting to make access must be a friend or member of the nominal class through which the access is made. See also message **`1061`**. **[11 section 11]**

1061 ***`String`*** **`member `*`Symbol`*` is not accessible through non-public inheritance`** -- There is an attempt to access a **`private`**, **`protected`** or **`public`** member (the text of the message indicates which kind as well as which member) of a class through a class derived from the original. There is an access violation (see message **`1060`** for the more common access violation) critically dependent on the fact

that the inheritance relationship is non-public. **[11 section 11.2]**

1062 **`template must be either a class or a function`** -- Following **`template < `*`arglist`*` >`** the parser expects to find either the token **`class`** or a function declaration or definition. **[10 section temp.decl]**

1063 **`Argument to copy constructor for class '`*`Symbol`*`' should be a reference`** -- A constructor for a class closely resembles a copy constructor. A copy constructor for class **`X`** is typically declared as:

```
X( const X &)
```

If you leave off the '**`&`**' then a copy constructor would be needed just to copy the argument into the copy constructor. This is a runaway recursion. **[11 section 12.1]**

1064 **`Template parameter list for template '`*`Symbol`*`' inconsistent with `*`Location`*** -- The **`template`** parameter list for a **`template`** function declaration or definition is inconsistent with that of a prior declaration or definition. **[10 section 14.temp.decl**

1065 **`Symbol '`*`Symbol`*`' not declared as "C" conflicts with `*`Location`*** -- A symbol previously declared as **`extern "C"`** in some other module is not declared as **`extern "C"`** in this module. This could be the source of very mysterious linker diagnostics since a name declared as **`extern "C"`** is not subject to the name mangling procedures that strictly C++ functions are subject to. **[11 section 7.4]**

1066 **`Symbol '`*`Symbol`*`' declared as "C" conflicts with `*`Location`*** -- A symbol is being declared as **`extern "C"`** and was not so declared in some other module. This could be the source of very mysterious linker diagnostics since a name declared as **`extern "C"`** is not subject to the name mangling procedures that strictly C++ functions are subject to. **[11 section 7.4]**

1067 **`invalid prototype for function '`*`Symbol`*`'`** -- Whenever **`operator delete`** or **`operator delete[]`** is defined, its first parameter must be declared as **`void *`**. For member functions, an optional second parameter may be **`size_t`**. **[10 section class.free]**

1068 **`Symbol '`*`Symbol`*`' can not be overloaded`** -- **`operator delete`** or **`operator delete[]`** can be redefined but not overloaded. There can be only one **`operator delete`** and one **`operator delete[]`** but neither of these can be overloaded. **[10 section class.free]**

1069 **`Symbol '`*`Name`*`' is not a base class of class '`*`Name`*`'`** -- Within a constructor initialization list, a name was found that did not correspond to either a direct base class of the class being defined or a member of the class.

1070 **No scope in which to find symbol '*Name*'** -- This could arise in an expression of the form **X::Y** where **X** does not represent a valid scope.

1071 **Constructors and destructors can not have return type** -- Constructors and destructors may not be declared with a return type, not even **void**. See **[ARM 11 section 12.1** and **12.4]**

1072 **Reference variable '*Symbol*' must be initialized** -- A reference variable must have an initializer at the point of declaration.

1073 **Insufficient number of template parameters; '*String*' assumed** -- A (**class**) template instantiation did not have a sufficient number of parameters. ***String*** indicates what the missing argument is presumed to be.

1074 **Expected a namespace identifier** -- In a declaration of the form:

```
namespace name = scoped-identifier
```

the ***scoped-identifier*** must identify a namespace.

1075 **Ambiguous reference to symbol '*Symbol*' and symbol '*Symbol*'** -- Two namespaces contain the same name. A reference to such a name could not be disambiguated. You must fully qualify this name in order to indicate the name intended.

1076 **Anonymous union assumed to be 'static'** -- Anonymous unions need to be declared **static**. This is because the names contained within are considered local to the module in which they are declared.

1077 **Could not evaluate default template parameter '*String*'** -- The evaluation of template parameters is deferred until needed. Thus:

```
template< class T = abc > class A { /* ... */ };
```

will be greeted with an Error **1077** only if an instantiation of **A** requires evaluation of the default argument and if that evaluation cannot be made. In that event **int** is assumed for type parameters and 0 is assumed for object parameters.

1078 **class '*Symbol*' should not have itself as a base class** -- The following situation will trigger this message.

```
class A : public A { };
```

You can't define **A** in terms of itself as there is no escape from the recursive plummet.

1079 **Could not find '>' or ',' to terminate template parameter at *Location*** -- The default value for a template parameter appears to be malformed. For example, suppose the user mistakenly substituted a '**]**' for a '**>**' producing the following:

```
template <class T = A< int ] >
    class X
        {
        };
```

This will cause PC-lint/FlexeLint to process to the end of the file looking (in vain) for the terminating pointy bracket. Not finding it will cause this message to be printed. Fortunately, the message will bear the ***Location*** of the malformed template.

1080 **Definition for class *'Name'* is not in scope** -- This message would be issued whenever a class definition is required but not available. For example:

```
Class X;        // declare class X
X *p;           // OK, no definition required
X a;            // Error 1080
```

1081 **Object parameter does not contain the address of a variable** -- A template argument that is passed to a pointer parameter is supposed to identify a symbol. The expression passed does not do so. For example

```
template< int *P > class A { ... };
int a[10];
A< a+2 > x;     // a+2 does not represent a symbol
```

1082 **Object parameter for a reference type should be an external symbol** -- A template argument that is passed to a reference parameter is supposed to identify an external symbol. The expression passed does not do so. For example

```
template< int &I > class A { ... };
int a[10];
A< a[2] > x;     // a[2] does not represent a symbol
```

See also message **1081**.

1083 **Ambiguous conversion between 2nd and 3rd operands of conditional operator** -- If the 2nd operand can be converted to match the type of the 3rd, and the 3rd operand can be converted to match the type of the 2nd, then the conditional expression is considered ill-formed.

1085 **Invalid definition of *'String'*** -- An attempt was made to define a member of a template before the template was defined. Example:

```
template<class T, class U> struct A
```

```
        {
        void
        };
template<class U, class T> void A<T,U>::f(){}  // Error 1085
```

In this case, the template argument list is out of order; T and U have been interchanged.

1086 **Compound literals may be used only in C99 programs** -- Compound literals are defined in C99 (ISO/IEC 9899:1999). However, some compilers allow the use of compound literals in C++. If you plan to port your code to another C++ compiler, then it may be worthwhile to heed this message; otherwise it may be safely suppressed with **-e1086**.

1087 **Previous declaration of *'Name'* (*Location*) is incompatible with *'Name'* (*Location*) which was introduced by the current using-declaration** -- A using-declaration such as:

```
using NS::name;
```

seems to be in error. It introduces a name that clashes with the name introduced earlier by another using-declaration. E.g.:

```
namespace N { int i;}
namespace Q { void i();}
using N::i;
using Q::i; // Error 1087 issued here.
```

1088 **A using-declaration must name a qualified-id** -- This error is issued when a using-declaration references a name without the **::** scope resolution operator; e.g.:

```
class A
    {
  protected:
    int n;
    };
class B : public A
    {
  public:
    using n; // Error 1088: should be 'using A::n;'
    };
```

See **[34]**, 7.3.3 **namespace.udecl**.

1089 **A using-declaration must not name a namespace** -- This error is issued when the rightmost part of the qualified-id in a using-declaration is the name of a namespace. E.g.:

```
namespace N
    {
    namespace Q
      {
      void g();
      }
    }
    void f()
        {
        using ::N::Q; // Error 1089
        Q::g();
        }
```

Instead, use a namespace-alias-definition:

```
namespace N
    {
    namespace Q
        {
        void g();
        }
    }
    void f()
        {
        namespace Q = ::N::Q; // OK
        Q::g(); // OK, calls ::N::Q::g().
        }
```

See **[35]**, Issue 460.

1090 **A using-declaration must not name a template-id** -- This error is issued when the rightmost part of the qualified-id in a using-declaration is a template-id. E.g.:

.

```
template<class T> class A
    {
  protected:
    template<class U> class B{};
    };

struct D : public A<int>
    {
  public:
    using A<int>::B<char*>; // Error 1090
    };

D::B<char*> bc;
```

Instead, refer to the template name without template arguments:

```
template<class T> class A
    {
  protected:
    template<class U> class B{};
    };

struct D : public A<int>
    {
  public:
    using A<int>::B; // OK
    };

D::B<char*> bc; // OK
```

See **[34]**, 7.3.3 **`namespace.udecl`**.

1091 ***'Name'*** **is not a base class of** ***'Name'*** -- This error is issued when the nested-name-specifier of the qualified-id in a using-declaration does not name a base class of the class containing the using-declaration; e.g.:

```
struct N
    {
    void f();
    }

class A
    {
  protected:
    void f();
    };
class B : A
    {
  public:
    using N::f; // Error 1091
    };
```

See **[35]**, Issue 400.

1092 **A using-declaration that names a class member must be a member-declaration** -- This error is issued when the nested-name-specifier of the qualified-id in a using-declaration names a class but the using-declaration does not appear where class members are declared. E.g.:

```
struct A
    {
    void f();
```

```
        };

    struct B : A
        {
        void g()
            {
            using A::f; // Error 1092
            }
        };
```

See **[34]**, 7.3.3 **`namespace.udecl`**.

1093 **`A pure specifier was given for function 'Symbol' which was not declared virtual`** -- A pure specifier (**`"= 0"`**) should not be placed on a function unless the function had been declared "**`virtual`**".

1094 **`Could not find ')' or ',' to terminate default function argument at Location`** -- A default function argument was found which did not seem to include any terminating tokens (the '**`,`**' separating arguments or '**`)`**' ending the function's argument list). Consequently, Lint continued scanning to the end of the file. ***`Location`*** indicates where the default argument began.

1095 **`Effective type 'Type' of non-type template parameter #Integer (corresponding to argument expression 'String') depends on an unspecialized parameter of this partial specialization`** -- The ISO C++ Standard says that "the type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization." [**34**], 14.5.4 temp.class.spec. Example:

```
// primary template:
template<class T, T N, class U> struct B;

// PS #1:
template<class U> struct B<int,257,U>; // Ok

// PS #2:
template<class U> struct B<bool,257,U>; // Ok, same as:
template<class U> struct B<bool,true,U>; // Ok (redeclaration of #2)

// PS #3:
template<class U> struct B<T,257,U>; // Error 1095 here
```

In PS #3, the value 257 is the 'specialized non-type argument' and its corresponding parameter is '**`N`**' whose type is **`T`** which was not made concrete. But in PS #1 and PS #2, **`T`** was given the concrete types '**`int`**' and '**`bool`**', respectively.

19.8 Additional Internal Errors

1200-1299 Some inconsistency or contradiction was discovered in the PC-lint/FlexeLint system. This may or may not be the result of a user error. This inconsistency should be brought to the attention of Gimpel Software.

19.9 C++ Warning Messages

1401 **member symbol '*Symbol*' (*Location*) not initialized by constructor** -- The indicated member symbol was not initialized by a constructor. Was this an oversight?

1402 **member '*Symbol*' (*Location*) not initialized** -- The indicated member symbol was not initialized before use. Either this is in a constructor where it is presumed that no members are pre-initialized or this is after a statement removing its initialization such as a **delete** or a **free**.

1403 **member '*Symbol*' (*Location*) not initialized** -- The indicated member symbol was not initialized before a point where its address is being passed to a constant pointer. This looks suspicious. Either this is in a constructor where it is presumed that no members are pre-initialized or this is after a statement removing its initialization such as a **delete** or a **free**.

1404 **deleting an object of type '*Symbol*' before type is defined** -- The following situation was detected:

```
class X; … X *p; … delete p;
```

That is, a placeholder declaration for a class is given and an object of that type is deleted before any definition is seen. This may or may not be followed by the actual class definition:

```
class X { … };
```

A **delete** before the class is defined is dangerous because, among other things, any **operator delete** that may be defined within the class could be ignored.

1405 **Header <typeinfo> must be included before typeid is used** -- According to Section 5.2.8 (para 6) of the C++ standard **[10]**, "If the header **<typeinfo>** (18.5.1) is not included prior to a use of **typeid**, the program is ill-formed." A **typeid** was found in the program but the required include was not.

1411 **Member with different signature hides virtual member '*Symbol*' (*Location*)** -- A member function has the same name as a virtual member of a

derived class but it has a different signature (different parameter list). This is legal but suspicious, because it looks as though the function would override the virtual function but doesn't. You should either adjust the parameters of the member so that the signatures conform or choose a different name. See also message **1511**.

1412 **Reference member '*Symbol*' is not initialized** -- A **class** member typed reference to **class** (or **struct** or **union**) is mentioned in a constructor initializer list. But the **class** (or **struct** or **union**) referenced has no constructor and so is never initialized.

1413 **function '*Symbol*' is returning a temporary via a reference** -- It appears that a function (identified as ***Symbol*** in the message) declared to return a reference is returning a temporary. The C++ standard (Section 12.2), in addressing the issue of binding temporary values to references, says "A temporary bound to the returned value in a function return statement ... persists until the function exits". Thus the information being returned is not guaranteed to last longer than the function being called.

It would probably be better to return by value rather than reference. Alternatively, you may return a static variable by reference. This will have validity at least until the next call upon the same function.

1414 **Assigning address of auto variable '*Symbol*' to member of this** -- The address of an auto variable was taken and assigned to a **this** member in a member function. For example:

```
struct A
    {
    char *x;
    void f()
        {
        char y[10];
        x = y;          // warning 1414
        }
    };
```

Here the address of **y** is being passed to member **x** but this is dangerous (if not ridiculous) since when the function returns the storage allocated for **y** is deallocated and the pointer could very easily harm something.

1415 **Pointer to non-POD class '*Name*' passed to function '*Symbol*' (*Context*)** -- A non-POD class is one which goes beyond containing just Plain Old Data (POD). In particular it may have private or protected data or it may have constructors or a destructor or a copy assignment. All of these things disqualify it from being a POD. A POD is fully defined in the C++ standard (Clause 9).

Some functions such as **memcpy**, **memcmp**, **memmove**, etc. are expected to be given only pointers to POD objects. The reason is that only POD objects have the property that they can be copied to an array of bytes and back again with a guarantee that they will retain their original value. (See Section 3.9 of the C++ standard **[34]**). See also Semantic **pod(*i*)**.

1416 **An uninitialized reference '*Symbol*' is being used to initialize reference '*Symbol*'** -- This message is usually issued when a reference to a member of a class is used to initialize a reference to another member of the same class before the first member was initialized. For example:

```
class C
    {
    int &n, &m;
    C( int &k ) : n(m), m(k) { /* ... */ }
    };
```

Here **m** is initialized properly to be identical to **k**. However, the initialization of **n**, taking place, as it does, before **m** is so initialized, is erroneous. It is undefined what location **n** will reference.

1417 **reference member '*Symbol*' not initialized by constructor initializer list** -- This message is issued when a reference data member of a class does not appear in a mem-initializer. For example, the following code will result in a Warning 1417 for symbol **m** since a mem-initializer is the only way that **m** can be reference initialized.

```
class C
    {
    int &n, &m;
    C( int &k ) : n(k) { /* ... */ }
    };
```

1501 **data member '*Symbol*' has zero size** -- A data member had zero size. It could be an array of zero length or a class with no data members. This is considered an error in C (Error **43**) but in C++ we give this warning. Check your code to make sure this is not an error. Some libraries employ clever templating, which will elicit this message. In such a case it is necessary for you to inhibit the message outright (using **-e1501**) or through a judicious use of **-esym(1501,...)**.

1502 **defined object '*Symbol*' of type *Name* has no non-static data members** -- A variable (***Symbol***) is being instantiated that belongs to a class (***Name***) that contains no data members (either directly or indirectly through inheritance). Note that this message can be suppressed using **-esym** of either the object name or the class name. **[11 section 9]**

1503 **`a tagged union is not anonymous`** -- A tagged union without a declarator appeared within a **`struct`**/**`union`** declaration. An anonymous **`union`** requires no tag. **[11 section 9.6]**

1504 **`useless struct declaration`** -- An untagged struct declaration appeared within a **`struct/union`** and has no declarator. It is not treated like an anonymous union. Was this intended?

1505 **`no access specifier provided, '`*`String`*`' assumed`** -- A base class specifier provides no access specifier (**`public`**, **`private`** or **`protected`**). An explicit access specifier is always recommended since the default behavior is often not what is expected. For example:

```
class A : B { int a; };
```

would make **`B`** a private base class by default.

```
class A : private B { int a; };
```

is preferred if that's what you want. **[11 section 11.1]**

1506 **`Call to virtual function '`*`Symbol`*`' within a constructor or destructor`** -- A call to a virtual function was found in a constructor or a destructor of a class. If this class is a base class of some other class (why else make a virtual call?), then the function called is not the overriding function of the derived class but rather the function associated with the base class. If you use an explicit scope operator this message will not be produced. **[20 section 9]**

1507 **`attempting to 'delete' an array`** -- The type of an object to be **`delete`**'d is usually a pointer. This is because operator **`new`** always returns a pointer and **`delete`** may only delete that allocated via **`new`**. Perhaps this is a programmer error attempting to delete a local or global array? **[19]**

1509 **`base class destructor for class '`*`Name`*`' is not virtual`** -- The indicated class is a base class for some derived class. It has a non-virtual destructor. Was this a mistake? It is conventional to virtualize destructors of base classes so that it is safe to **`delete`** a base class pointer. **[19]**

1510 **`base class '`*`Name`*`' has no destructor`** -- The indicated class is a base class for some derived class that has a destructor. The base class does not have a destructor. Is this a mistake? The difficulty that you may encounter is this; if you represent (and manipulate) a heterogeneous collection of possibly derived objects via a pointer to the base class then you will need a virtual base class destructor to invoke the derived class destructor. **[13 section 4]**

1511 **`Member hides non-virtual member '`*`Symbol`*`' (`*`Location`*`)`** -- The named

member of a derived class hides a similarly named member of a base class. Moreover, the base class member is not virtual. Is this a mistake? Was the base member supposed to have been declared **`virtual`**? By unnecessarily using the same name, confusion could be created.

1512 **`destructor for base class '`*`Symbol`*`' (`*`Location`*`) is not virtual`** -- In a final pass through all the classes, we have found a class (named in the message) that is the base class of a derivation and has a destructor but the destructor is not virtual. It is conventional for inherited classes to have virtual destructors so that it is safe to '**`delete`**' a pointer to a base class. **[19]**

1513 **`storage class ignored`** -- A storage class (one of **`auto`**, **`extern`**, or **`register`**) was found within a class definition. The only storage classes that are significant when declaring members are **`static`** and **`typedef`**. **[11 section 9.2]**

1514 **`Creating temporary to copy '`*`Type`*`' to '`*`Type`*`' (context: `*`Context`*`)`** -- A temporary was created in order to initialize (or pass a value to or return a value to) a reference. This is suspect because any modification to the value will be a modification of this temporary. This message is not issued when initializing a **`const`** reference. **[11 section 12.2]**

1515 **`Default constructor not available for member '`*`Symbol`*`'`** -- A member of a class was found that had a type for which a constructor was defined but for which a default constructor (one with no arguments) was not defined.

1516 **`Member declaration hides inherited member '`*`Symbol`*`' (`*`Location`*`)`** -- A data member of a class happens to have the same name as a member of a base class. Was this deliberate? Identical names can cause confusion. To inhibit this message for a particular symbol or for an identifiable set of symbols use **`-esym()`**.

1520 **`Multiple assignment operators for class '`*`Symbol`*`'`** -- More than one assignment operator has been declared for a given class. For example, for class **`X`** there may have been declared:

```
void operator=(X);
void operator=(X) const;
```

Which is to be used for assignment?

1521 **`Multiple copy constructors for class '`*`Symbol`*`'`** -- For a given class, more than one function was declared that could serve as a copy constructor. Typically this means that you declared both **`X( X& )`** and **`X( const X& )`** for the same class. This is probably a mistake.

1522 **`Symbol '`*`Symbol`*`' is an array of empty objects`** -- An array (***`Symbol`***) is being allocated. Each member of the array appears to be empty. Although this is legal,

it could be the result of human error. If this is a deliberate policy, inhibit the message, either globally, or for this ***Symbol***.

1524 **new in constructor for class '*Name*' which has no explicit destructor** -- A call to **new** has been found in a constructor for a class for which no explicit destructor has been declared. A destructor was expected because how else can the storage be freed? **[10 section class.free]**

1526 **Member function '*Symbol*' (*Location*) not defined** -- A member function (named in the message) of a non-library class was not defined. This message is suppressed for unit checkout (**-u** option).

1527 **static member '*Symbol*' (*Location*) not defined** -- A static data member (named in the message) of a non-library class was not defined. In addition to its declaration within the class, it must be defined in some module.

1528 **call to *String* does not match function template *'String'*** -- The first ***String*** of the message designates an actual function call that appeared to be the invocation of the template function identified by the second ***String***. No match could be made between the arguments of the call and the template parameters.

1529 **Symbol '*Symbol*' not first checking for assignment to this** -- The assignment operator does not appear to be checking for assignment of the value of a variable to itself (assignment to **this**). Specifically PC-lint/FlexeLint is looking for one of:

```
if( &arg == this )
if( &arg != this )
if( this == &arg )
if( this != &arg )
```

as the first statement of the function.

It is important to check for a self assignment so as to know whether the old value should be subject to a delete operation. This is often overlooked by a class designer since it is counter-intuitive to assign to oneself. But through the magic of aliasing (pointers, references, function arguments) it is possible for an unsuspecting programmer to stumble into a disguised self-assignment. **[12, Item 17]**

If you are currently using the following test

```
if( arg == *this)
```

we recommend you replace this with the more efficient:

```
if( &arg == this || arg == *this)
```

1531 **Symbol '*Symbol*' (*Location*) should have compared argument against sizeof(class)** -- This warning is given for either **operator new** or **operator delete** when defined as member functions of a class that is the base class of a derivation. In this case you can't be certain of the size of allocation and therefore your allocation functions should test the size parameter for equality to the **sizeof** the class. See Elective Note **1921** for more details.

1532 **Symbol '*Symbol*' not checking argument for NULL** -- This message is given for a function **operator delete** which is a member function of a class that does not have a destructor. It should check for NULL because **delete p** where **p** has the NULL value will be passed in to it. See also **1922**.

1533 **Repeated friend declaration for symbol '*Symbol*'** -- A **friend** declaration for a particular symbol (class or function) was repeated in the same class. Usually this is a harmless redundancy.

1534 **static variable '*Symbol*' found within inline function in header** -- A static variable (***Symbol***) was found within an inline function within a header file. This can be a source of error since the static variable will not retain the same value across multiple modules. Rather each module will retain its own version of the variable. If multiple modules need to use the function then have the function refer to an external variable rather than a static variable. Conversely if only one module needs to use the function then place the definition of the function within the module that requires it. **[23, Item 26]**

1535 **Exposing low access data through member '*Symbol*'** -- A member function is returning an address being held by the indicated member symbol (presumably a pointer). The member's access (such as **private** or **protected**) is lower than the access of the function returning the address.

1536 **Exposing low access member '*Symbol*'** -- A member function is returning the non-const address of a member either directly or via a reference. Moreover, the member's access (such as **private** or **protected**) is lower than the access of the function returning the address. For example:

```
class X
    {
   private:
    int a;
   public:
    int *f() { return &a; }
    };
```

This looks like a breach of the access system **[12, Item 30]**. You may lower the access rights of the function, raise the accessibility of the member or make the return value a

const pointer or reference. In the above example you could change the function to:

```
const int *f() { return &a; }
```

1537 **const function returns pointer data member '*Symbol*'** -- A **const** function is behaving suspiciously. It is returning a pointer data member (or equivalently a pointer to data that is pointed to by a data member). For example,

```
class X
    {
    int *p;
    int *f() const { return p; }
    };
```

Since **f** is supposedly **const** and since **p** is presumptively pointing to data that is logically part of **class X** we certainly have the potential for a security breach. Either return a pointer to **const** or remove the **const** modifier to the function. **[12, Item 29]**

Note, if a **const** function returns the address of a data member then a **605** (capability increase) is issued.

1538 **base class '*Name*' absent from initializer list for copy constructor** -- The indicated base class did not appear in the initializer list for a copy constructor. Was this an oversight? If the initializer list does not contain an initializer for a base class, the default constructor is used for the base class. This is not normally appropriate for a copy constructor. The following is more typical:

```
class B { ... };
class D : public B
    {
    D( const D &arg ) : B( arg ) { ... }
    ...
    };
```

1539 **member '*Symbol*' (*Location*) not assigned by assignment operator** -- The indicated ***Symbol*** was not assigned by an assignment operator. Was this an oversight? It is not strictly necessary to initialize all members in an assignment operator because the '**this**' class is presumably already initialized. But it is easy to overlook the assignment of individual members. It is also easy to overlook your responsibility to assign base class members. This is not done for you automatically. **[12, Item 16]**

The message is not given for **const** members or reference members. If you have a member that is deliberately not initialized you may suppress the message for that member only using **-esym**.

1540 **pointer member '*Symbol*' (*Location*) neither freed nor zero'ed by destructor** -- The indicated member is a non-static pointer member of a class that

was apparently not freed by the class destructor. Was this an oversight? By freeing, we mean either a call to the **free()** function or use of the **delete** operator. If the pointer is intended only to point to static information during its lifetime then, of course, it never should be freed. In that case you should signal closure by assigning it the NULL pointer (**0**).

1541 **member '*Symbol*' (*Location*) possibly not initialized by constructor** -- The indicated member symbol may not have been initialized by a constructor. Was this an oversight? Some of the paths that the constructor takes do initialize the member. See Section 10.1 Initialization Tracking

1542 **member '*Symbol*' (*Location*) possibly not initialized** -- The indicated member symbol may not have been initialized before use. Either this is in a constructor where it is presumed that no members are pre-initialized or this is after a statement removing its initialization such as a **delete** or a **free**. See Section 10.1 Initialization Tracking

1543 **member '*Symbol*' (*Location*) possibly not initialized** -- The indicated member symbol may not have been initialized before a point where its address is being passed to a constant pointer. This looks suspicious. Either this is in a constructor where it is presumed that no members are pre-initialized or this is after a statement removing its initialization such as a **delete** or a **free**.

1544 **value of variable '*Symbol*' (*Location*) indeterminate (order of initialization)** -- A variable (identified by ***Symbol***) was used in the run-time initialization of a static variable. However this variable itself was initialized at run-time. Since the order of initialization cannot be predicted this is the source of possible error.

Whereas addresses are completely known at initialization time, values may not be. Whether the value or merely the address of a variable is used in the initialization of a second variable is not an easy thing to determine when an argument is passed by reference or via pointer. For example,

```
class X
    {
    X( const X & );
    };

extern X x1;
X x2 = x1;
X x1 = x2;
```

It is theoretically possible, but unlikely, that the constructor **X()** is interested only in the address of its argument and not its current value. If so, it only means you will be getting a spurious report, which you can suppress based on variable name. However, if the **const** is missing when passing a reference parameter (or a pointer parameter) then we

cannot easily assume that values are being used. In this case no report will be issued. The moral is that if you want to get the checking implied by this message you should make your constructor reference arguments **const**.

1545 **value of variable '*Symbol*' used previously to initialize variable '*Symbol*' (*Location*)** -- A variable identified by ***Symbol*** was used previously to initialize some other variable. This variable is now itself being initialized with run-time code. The order of these initializations cannot be predicted. See also message **1544**.

1546 **throw() called within destructor '*Symbol*'** -- The body of a destructor (signature provided within the message) contains a **throw** not within a **try** block. This is dangerous because destructors are themselves triggered by exceptions in sometimes unpredictable ways. The result can be a perpetual loop. **[23, Item 11]**

1547 **Assignment of array to pointer to base class (*Context*)** -- An assignment from an array of a derived class to a pointer to a base class was detected. For example:

```
class B { };
class D : public B {};
D a[10];
B *p = a;      // Warning 1547
B *q = &a[0];  // OK
```

In this example **p** is being assigned the address of the first element of an array. This is fraught with danger since access to any element other than the zeroeth must be considered an error (we presume that **B** and **D** actually have or have the potential to have different sizes). **[23, Item 3]**

We do not warn about the assignment to **q** because it appears that the programmer realizes the situation and wishes to confine **q** to the base object of the zeroeth element of **a** only. As a further precaution against inappropriate array access, out of bounds warnings are issued for subsequent references to **p[1]** and **q[1]**.

1548 **Exception specification for '*Symbol*' conflicts with *Location*** -- The exception specification of a function begins with the keyword '**throw**' and follows the prototype. Two declarations were found for the same function with inconsistent exception specifications.

1549 **Exception thrown for function '*Symbol*' not declared to throw** -- An exception was thrown (i.e., a **throw** was detected) within a function and not within a **try** block; moreover the function contains a **throw** specification but the exception thrown was not on the list. If you provide an exception specification, include all the exception types you potentially will throw. **[23, Item 14]**

1550 **exception '*Name*' thrown by function '*Symbol*' is not on throw-list of function '*Symbol*'** -- A function was called (first ***Symbol***) which was declared as potentially throwing an exception. The call was not made from within a **try** block and the function making the call had an exception specification. Either add the exception to the list, or place the call inside a **try** block and **catch** the **throw**. **[23, Item 14]**.

1551 **function '*Symbol*' may throw an exception in destructor '*Symbol*'** -- A call to a function (name given by the first ***Symbol***) was made from within a destructor. The function was declared as potentially throwing an exception. Such exceptions need to be caught within a **try** block because destructors should never throw exceptions. **[23, Item 11]**.

1552 **Converting pointer to array-of-derived to pointer to base** -- This warning is similar to Warning **1547** and is sometimes given in conjunction with it. It uses value tracking to determine that an array (that could be dynamically allocated) is being assigned to a base class pointer.

For example,

```
Derived *d = new Derived[10];
Base *b;
b = d;      // Warning 1552
b = &d[0];  // OK
```

[23, Item 3] Also, see the article by Mark Nelson (Bug++ of the Month, Windows Developer's Journal, May 1997, pp. 43-44).

1553 **struct '*Symbol*' declared as extern "C" contains C++ substructure '*Symbol*' (*Location*)** -- A C++ substructure was found in a structure or class declared as **extern "C"**. Was this intended?

1554 **Direct pointer copy of member '*Symbol*' within copy constructor: '*Symbol*'** -- In a copy constructor a pointer was merely copied rather than recreated with new storage. This can create a situation where two objects have the same data and this, in turn, causes problems when these objects are deleted or modified. For example, the following class will draw this warning:

```
class X
    {
    char *p;
    X( const X & x )
        { p = x.p; }
    ...
    };
```

Here, member **p** is expected to be recreated using **new** or some variant.

1555 **Direct pointer copy of member '*Symbol*' within copy assignment operator: '*Symbol*'** -- In a copy assignment operator a pointer was merely copied rather than recreated with new storage. This can create a situation where two objects have the same data and this, in turn, causes problems when these objects are deleted or modified. For example, the following class will draw this warning:

```
class X
    {
    char *p;
    X& operator=( const X & x )
        { p = x.p; }
    ...
    };
```

Here, member **p** is expected to be recreated using **new** or some variant.

1556 **'new Type(integer)' is suspicious** -- A **new** expression had the form **new *T*(*Integer*)** where type ***T*** has no constructor. For example:

```
new int(10);
```

will draw this warning. The expression allocates an area of storage large enough to hold one integer. It then initializes that integer to the value **10**. Could this have been a botched attempt to allocate an array of 10 integers? Even if it was a deliberate attempt to allocate and initialize a single integer, a casual inspection of the code could easily lead a reader astray.

The warning is only given when the type ***T*** has no constructor. If ***T*** has a constructor then either a syntactic error will result because no constructor matches the argument or a match will be found. In the latter case no warning will or should be issued.

1557 **const member '*Symbol*' is not initialized** -- A class member typed **const** class (or struct or union) is mentioned in a constructor initializer list. But the class (or struct or union) referenced has no constructor and hence the member is not initialized. See also message **1769**.

1558 **'virtual' coupled with 'inline' is an unusual combination** -- The function declared both **virtual** and **inline** has been detected. An example of such a situation is as follows:

```
class C
    {
    virtual inline void f();   // Warning 1558
    };
```

Virtual functions by their nature require an address and so inlining such a function seems contradictory. We recommend that the **inline** function specifier be removed.

1559 **Uncaught exception '*Name*' may be thrown in destructor '*Symbol*'** -- The named exception occurred within a **try** block and was either not caught by any handler or was caught but then thrown from the handler. Destructors should normally not throw exceptions. **[23, Item 11]**

1560 **Uncaught exception '*Name*' not on throw-list of function '*Symbol*'** -- A direct or indirect **throw** of the named exception occurred within a **try** block and was either not caught by any handler or was rethrown by the handler. Moreover, the function has an exception specification and the uncaught exception is not on the list. Note that a function that fails to declare a list of thrown exceptions is assumed to potentially throw any exception.

1561 **Reference initialization causes loss of const/volatile integrity (*Context*)** -- A reference initialization is resulting in a capability gain that can cause a loss of const or volatile integrity.

Typically the message is given on initializing a non-const reference with a **const**. For example:

```
void f( int &x );
const int n = 0;
...
f(n);
```

Here, function **f()** could assign a value to its argument and thereby modify **n**, which is declared to be **const**.

The message can also be issued when a pointer is initialized. Consider the following example.

```
void h( const int *&q );
int *p;
...
h(p);
```

It might seem that passing a regular (i.e., non-const) pointer to a **const int *** could cause no harm. That would be correct if it were not for the reference. If function **h()** were to assign a pointer to **const** to its parameter **q** then upon return from the call, **p** could be used to modify const data.

There are many subtle cases that can boggle the mind. See the commentary to Message **605**.

1562 **Exception specification for '*Symbol*' is not a subset of '*Symbol*' (*Location*)** -- The first ***symbol*** is that of an overriding virtual function for the second ***symbol***. The exception specification for the first was found not to be a subset of the second. For example, it may be reasonable to have:

```
struct B   { virtual void f() throw(B); };
struct D:B { virtual void f() throw(D); };
```

Here, although the exception specification is not identical, the exception **D** is considered a subset of the base class **B**.

It would not be reasonable for **D::f()** to throw an exception outside the range of those thrown by **B::f()** because in general the compiler will only see calls to **B::f()** and it should be possible for the compiler to deduce what exceptions could be thrown by examining the static call.

1563 **Suspicious third argument to ?: operator** -- The third argument to **?:** contained an unparenthesized assignment operator such as

```
p ? a : b = 1
```

If this is what was intended you should parenthesize the third argument as in:

```
p ? a : (b = 1)
```

Not only is the original form difficult to read but C, as opposed to C++, would parse this as:

```
(p ? a : b) = 1
```

1564 **Assigning a non-zero-one constant to a bool** -- The following looks suspicious.

```
bool a = 34;
```

Although there is an implicit conversion from integral to **bool** and assigning an integer variable to a **bool** to obtain its Boolean meaning is legitimate, assigning an integer such as this looks suspicious. As the message suggests, the warning is not given if the value assigned is either **0** or **1**. An Elective Note would be raised in that instance.

1565 **member '*Symbol*' (*Location*) not assigned by initializer function** -- A function dubbed '**initializer**' by a **-sem** option is not initializing (i.e., assigning to) every data member of a class. Reference members and **const** members theoretically can be initialized only via the constructor so that these members are not candidates for this message.

1566 **member '*Symbol*' (*Location*) might have been initialized by a separate function but no '-sem(*Name*,initializer)' was seen** -- A class data member (whose name and location are indicated in the message) was not directly initialized by a constructor. It may have been initialized by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint/FlexeLint that the separately called function is in fact an '**initializer**'. For example:

```
class A {
       int a:
    public:
       void f();
       A() { f(); }
       };
```

Here **f()** is presumably serving as an initializer for the constructor **A::A()**. To inform PC-lint/FlexeLint of this situation, use the option:

```
-sem( A::f, initializer )
```

This will suppress Warning **1566** for any constructor of **class A** that calls **A::f**.

1567 **Initialization of variable '*Symbol*' (*Location*) is indeterminate as it uses variable '*Symbol*' through calls: '*String*'** -- A variable was dynamically initialized using an expression that contained a call to a function and that function referenced a variable that was also dynamically initialized and was in some other module. For example:

```
a.cpp:                  b.cpp:

    int g(void);            int f(void);
    int y = g();            int x = f();
    int f() { return y; }
```

The initialization of both **x** and **y** are dynamic. Although the order of dynamic initialization within a module is pre-ordained the order in which modules are initialized is not. Therefore it is perfectly possible for **b.cpp** to be initialized before **a.cpp**. Thus when the call is made upon function **f()** to initialize **x**, variable **y** may not yet have been initialized.

1568 **Variable '*Symbol*' (*Location*) accesses variable '*Symbol*' before the latter is initialized through calls: '*String*'** -- A variable was dynamically initialized using an expression that contained a call to a function and that function referenced a variable that was also dynamically initialized but later in the module. For example:

```
    int g(void);
```

```
int f(void);
int x = f();
int y = g();
int f() { return y; }
```

The initialization of both **x** and **y** are dynamic. The order of dynamic initialization within a module is in the order in which the initialization is specified. Thus when the call is made upon function **f()** to initialize **x**, variable **y** will not yet have been initialized.

1569 **Initializing a reference with a temporary** -- A reference was initialized with a temporary. For example:

```
int f( int );
const int &x = f(3);
```

The expression **f(3)** returns an **int** which is placed in a temporary location. The life of the temporary is not guaranteed to extend beyond the declaration itself. Subsequent use of reference **x** is problematic. The same principle applies when initializing a reference member of a class. Consider:

```
struct A { int &n;  A() : n(3) {} };
```

The constructor **A()** contains an initializer list within which it initializes **n**. But **n** will be bound to a temporary created by the compiler to hold the value **3**. The lifetime of this temporary is limited and will not extend through the life of the object created by the constructor.

1570 **Initializing a reference class member with an auto variable *'Symbol'*** -- In a constructor initializer, a reference class member is being initialized to bind to an **auto** variable. Consider:

```
class X { int &n; X(int k) :n(k) {} };
```

In this example member **n** is being bound to variable **k** which, although a parameter, is nonetheless placed into **auto** storage. But the lifetime of **k** is only the duration of the call to the constructor, whereas the lifetime of **n** is the lifetime of the class object constructed.

1571 **Returning an auto variable *'Symbol'* via a reference type** -- A function that is declared to return a reference is returning an **auto** variable (that is not itself a reference). The **auto** variable is not guaranteed to exist beyond the lifetime of the function. This can result in unreliable and unpredictable behavior.

1572 **Initializing a static reference variable with an auto variable *'Symbol'*** -- A **static** variable has a lifetime that will exceed that of the **auto**

variable that it has been bound to. Consider

```
void f( int n ) { static int& r = n; ... }
```

The reference **r** will be permanently bound to an **auto** variable **n**. The lifetime of **n** will not extend beyond the life of the function. On the second and subsequent calls to function **f** the static variable **r** will be bound to a non-existent entity.

1573 **Generic function template *'Symbol'* declared in namespace associated with type *'Symbol'* (*Location*)** -- When a class (or union or enum) is declared within a namespace that namespace is said to be associated with the type. A Generic function template is any that has as parameters only intrinsic types or plain template arguments possibly adorned with reference or const or volatile qualification. Consider

```
namespace X
     {
     template< class T >
          void f ( int, const T& );    // Generic
     class A();                        // Warning 1573
     }
```

A call to function **f** that contained an argument of type **X::A** would, by ADL (Argument Dependent Lookup), need to also consider function **X::f** even though this function was not in the scope of the call.

Some designers adopt the strategy of embedding the class within a sub namespace and employing a **using** declaration to make it available to users of the original namespace. For example:

```
namespace X
     {
     template< class T >
          void f( int, const T& );     // Generic
     namespace X1
          {
          class A{};                   // No Warning
          }
     using X1::A;
     }
```

Now an argument of type **X::A** will not automatically trigger a consideration of **X::f**.

1576 **Explicit specialization does not occur in the same file as corresponding function template *'Symbol'* (*Location*)** -- An explicit specialization of a function template was found to be declared in a file other than the one in which the corresponding function template is declared. Two identical calls in two

different modules on the same function template could then have two differing interpretations based on the inclusion of header files. The result is undefined behavior.

As if this wasn't enough, if the explicit specialization could match two separate function templates then the result you obtain could depend on which function templates are in scope.

See also the next message.

1577 **Partial or explicit specialization does not occur in the same file as primary template '*Symbol*' (*Location*)** -- There is a danger in declaring an explicit specialization or a partial specialization in a file other than that which holds the primary class template. The reason is that a given implicit specialization will differ depending on what headers it sees. It can easily differ from module to module and undefined behavior can be the result.

See also Warning **1576** which diagnoses a similar problem with function templates.

1578 **Pointer member '*Symbol*' (*Location*) neither freed nor zeroed by cleanup function** -- The indicated member is a non-static data member of a class that was apparently not cleared by a function that had previously been given the **cleanup** semantic. By clearing we mean that the pointer was either zeroed or the storage associated with the pointer released via the **free** function or its semantic equivalent or some form of **delete**. See also Warning **1540**.

1579 **Pointer member '*Symbol*' (*Location*) might have been freed by a separate function but no '-sem(*Name*,cleanup)' was seen** -- A class data member (whose name and location are indicated in the message) was not directly freed by the class destructor. There was a chance that it was cleared by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint/FlexeLint that the separately called function is in fact a '**cleanup**' function. For example:

```
class A {
        int *p;
    public:
        void release_ptrs();
        ~A() { release_ptrs(); }
        };
```

Here **release_ptrs()** is presumably serving as a **cleanup** function for the destructor **~A::A()**. To inform PC-lint/FlexeLint of this situation, use the option:

```
-sem( A::release_ptrs, cleanup )
```

A separate message (Warning **1578**) will be issued if the **cleanup** function fails to

clear all pointers. See also Warning **1566**.

19.10 C++ Informational Messages

1701 **redundant access-specifier '*String*'** -- The given access specifier (one of **public**, **private** or **protected**) has been repeated. **[11 section 11.1]**

1702 **operator '*Name*' is both an ordinary function '*String*' and a member function '*String*'** -- In attempting to resolve the definition of an operator it was found that the same operator was declared as both a member function and a non-member function. Was this intended? Symmetric binary operators (such as '**+**', '**-**', '**==**', '**>**', etc.) are usually defined external to a class definition so that they can support non-objects on the left hand side. **[11 section 13.4.2]**

1703 **Function '*Name*' arbitrarily selected. Refer to Error '*Integer*'** -- This informational message is given with error numbers **1023**, **1024**, **1025** and **1026**. These are issued when an error is encountered during the overload resolution process, and is issued merely to indicate which function was arbitrarily selected. **[11 section 13.2]**

1704 **Constructor '*Symbol*' has private access specification** -- A **private** constructor is legal and has its uses but can also result in messages that are difficult to interpret. If you use **private** constructors as a programming technique then you may suppress this message with a **-e1704**. But it's probably better to suppress this on a constructor by constructor basis using **-esym**.

1705 **static class member may be accessed by the scoping operator** -- A static class member was accessed using a class object and **->** or **.** notation. For example:

```
s.member
```

or

```
p->member
```

But an instance of the object is not necessary. It could just as easily have been referenced as:

```
X::member
```

where **X** is the class name. **[10 section class.static]**

1706 **Declaration with scope operator is unusual within a class** -- Class members within a class are not normally declared with the scope operator. For example:

```
class X  {  int X::n; …
```

will elicit this message. If the (redundant) class specification (**X::**) were replaced by some different class specification and the declaration was not **friend** an error (**1040**) would be issued. **[11 section 9.2]**

1707 **static assumed for *String*** -- **operator new()** and **operator delete()**, when declared as member functions, should be declared as **static**. They do not operate on an object instantiation (implied **this** pointer). **[11 section 12.5]**

1708 **typedef '*Symbol* not declared as "C" conflicts with *Location*** -- A **typedef** symbol previously declared as **extern "C"** was not so declared at the current location. This is not considered as serious a situation as is indicated by message **1065**, which is given for external function and variable names. If this is your programming style, you may suppress this message.

1709 **typedef '*Symbol*' declared as "C" conflicts with *Location*** -- A **typedef** symbol was previously not declared as **extern "C"** but is so declared at the current location. This is not considered as serious a situation as is indicated by message **1066**, which is given for external function and variable names. If this is your programming style, you may suppress this message.

1710 **An implicit '*typename*' was assumed** -- This message is issued when the standard requires the use of '***typename***' to disambiguate the syntax within a template where it may not be clear that a name is the name of a type or some non-type. (See **C++ Standard [10], Section temp.res, Para 2**). Consider:

```
template< class T > class A
    {
    T::N x;    // Info 1710
    };
```

Many compilers will accept this construct since the only interpretation consistent with valid syntax is that **T::N** represents a type. (But if the '**x**' weren't there it would be taken as an access declaration and more frequently would be a non-type)

1711 **class '*Symbol*' (*Location*) has a virtual function but is not inherited** -- The given class has a virtual function but is not the base class of any derivation. Was this a mistake? There is no advantage to making member functions virtual unless their class is the base of a derivation tree. In fact, there is a disadvantage because there is a time and space penalty for virtual functions. This message is not given for library classes and is suppressed for unit checkout. **[13 section 4]**

1712 **default constructor not defined for class '*Name*'** -- A **class** was defined with one or more constructors but none of these could be used as a (0 argument) default constructor. Is this an omission? The default constructor is used in declarations

and for **new** when no explicit initialization is given. It is also used when the class is a base class and no mem-initializer is given. It is used for arrays as well. A default constructor should therefore be omitted only for good reason. If you have such a good reason for **class X** you can employ option **-esym(1712,X)**. **[19]**

1713 **Parentheses have inconsistent interpretation** -- An expression of the form:

```
new T()
```

is supposed to produce a default initialized allocation of type **T**. If **T** is a POD type (Plain Old Data type) it is supposed to be initialized to **0**. Since this change was made relatively late in the draft leading to the standard, many compilers do not yet support this construct. If your compiler does support the construct and you have no intention of porting your application to any other compiler, suppress this message. Alternatively, code this as:

```
new T
```

and initialize the result explicitly.

1714 **Member function '*Symbol*' (*Location*) not referenced** -- A member function was not referenced. This message is automatically suppressed for unit checkout (**-u**) and for members of a library class.

1715 **static member '*Symbol*' (*Location*) not referenced** -- A static data member of a class was not referenced. This message is automatically suppressed for unit checkout (**-u**) and for members of a library class.

1716 **Virtual member function '*Symbol*' (*Location*) not referenced** -- A virtual member function was apparently not referenced. Not only was the function itself not referenced but the function or functions that it overrides were not referenced either. The message is not given if the member function itself or any member function that it overrides is a library member function. This is because the original virtual function may be called implicitly by the library.

This message is suppressed for unit checkout (**-u**).

1717 **empty prototype for function declaration, assumed '(void)'** -- An empty prototype, as in:

```
void  f();
```

has a different meaning in C than in C++. In C it says nothing about the arguments of the function; in C++, it says there are no arguments. This message is not given for member function declarations or for function definitions. Rather, weaker Elective Notes

(**1917** and **1918**) are given. This is because the chance of ambiguity does not exist in those cases. **[11 section 8.2.5]**

1718 **expression within brackets ignored** -- In the expression:

```
delete  [ expression ]  p
```

the ***expression*** is ignored. The ***expression*** is a vestige of an earlier time when this information provided a count of the number of items in the array being released. Note that empty square brackets are considered necessary for deleting an array. This is a complaint directed toward the expression within the brackets not the brackets themselves. **[11 section 5.3.4]**

1719 **assignment operator for class '*Symbol*' has non-reference parameter** -- The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not a reference then your program is subject to implicit function calls and less efficient operation. **[11 section 13.4.3]**

1720 **assignment operator for class '*Symbol*' has non-const parameter** -- The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not **const** then your program will not be diagnosed as completely as it might otherwise be. **[11 section 13.4.3]**

1721 **operator =() for class '*Symbol*' is not assignment operator** -- The assignment operator for a class has the form:

```
X& operator =(const X &)
```

A member function whose name is **operator =**, but does not have that form, is not an assignment operator. This could be a source of subtle confusion for a program reader. If this is not an error you may selectively suppress this message for the given class. **[11 section 13.4.3]**

1722 **assignment operator for class '*Symbol*' does not return a reference to class** -- The typical assignment operator for a class **X** is of the form:

```
X& operator =(const X &);
```

The reason for returning a reference to class is to support multiple assignment as in:

```
a = b = c
```

[11 section 13.4.3]

1724 **`Argument to copy constructor for class '`*`Symbol`*`' should be a const reference`** -- A copy constructor for class **`X`** is typically declared as:

```
X( const X & );
```

If you leave off the '**`const`**' then some diagnostics will not be possible. **[19]**

1725 **`class member '`*`Symbol`*`' is a reference`** -- There are a number of subtle difficulties with reference data members. If a class containing a reference is assigned, the default assignment operator will presumably copy the raw underlying pointer. This violates the principle that a reference's underlying pointer, once established, is never modified. Some compilers protect against this eventuality by refusing to create a default assignment operator for classes containing references. Similar remarks can be made about copy constructors. If you are careful about how you design your copy constructors and assignment operators, then references within classes can be a useful programming technique. They should not, however, be employed casually. **[21 section 2.1.3]**

1726 **`taking address of overloaded function name '`*`Symbol`*`'`** -- A reference is being made to an overloaded function without an immediately following '**`(`**'. Thus there is no argument list to distinguish the function intended. Resolution of the overloaded name can only be made by analyzing the destination. Is this what the programmer intended? **[11 section 13.3]**

1727 **`inline '`*`Symbol`*`' not previously defined inline at (`*`Location`*`)`** -- A function declared or defined inline was not previously declared inline. Was this intended? If this is your standard practice then suppress this message. **[11 section 9.3.2]**

1728 **`Symbol '`*`Symbol`*`' was previously defined inline at (`*`Location`*`)`** -- A function was previously declared or defined inline. The **`inline`** modifier is absent from the current declaration or definition. Was this intended? If this is your standard practice then suppress this message. **[11 section 9.3.2]**

1729 **`Initializer inversion detected for member '`*`Symbol`*`'`** -- In a constructor initializer the order of evaluation is determined by the member order not the order in which the initializers are given. At least one of the initializers was given out of order. Was there a reason for this? Did the programmer think that by changing the order that he/she would affect the order of evaluation? Place the initializers in the order of their occurrence within the class so that there can be no mistaken assumptions. **[12, Item 13]**

1730 **`class/struct inconsistency for symbol '`*`Symbol`*`' (conflicts with`**

***Location*)** -- An object is declared both with the keyword **class** and with the keyword **struct**. Though this is legal it is suspect. **[11 section 7.1.6]**

1732 **new in constructor for class '*Name*' which has no assignment operator** -- Within a constructor for the cited class, there appeared a **new**. However, no assignment operator was declared for this class. Presumably some class member (or members) points to dynamically allocated memory. Such memory is not treated properly by the default assignment operator. Normally a custom assignment operator would be needed. Thus, if **x** and **y** are both of type ***Name***

```
x = y;
```

will result in pointer duplication. A later **delete** would create chaos. **[12, Item 11]**

1733 **new in constructor for class '*Name*' which has no copy constructor** -- Within a constructor for the cited class, there appeared a **new**. However, no copy constructor was declared for this class. Presumably, because of the **new**, some class member (or members) points to dynamically allocated memory. Such memory is not treated properly by the default copy constructor. Normally a custom copy constructor would be needed. **[12, Item 11]**

1734 **Had difficulty compiling template function: '*Symbol*'** -- At template wrap-up time where there is an attempt to 'compile' each template function according to arguments provided, the cited function could not be processed fully. The difficulty may be the result of syntax errors cited earlier and if these errors are repaired then this message should go away.

1735 **Virtual function '*Symbol*' has default parameter** -- A virtual function was detected with a default parameter. For example:

```
class B
    {
    virtual void f( int n = 5 );
    ...
    };
```

The difficulty is that every virtual function **f** overriding this virtual function must contain a default parameter and its default parameter must be identical to that shown above. If this is not done, no warnings are issued but behavior may have surprising effects. This is because when **f()** is called through a base class pointer (or reference) the function is determined from the actual type (the dynamic type) and the default argument is determined from the nominal type (the static type). **[12, Item 38]**.

1736 **Redundant access specifier (*String*)** -- An access specifier (one of **public**, **private**, or **protected** as shown in ***String***) is redundant. That is, the explicitly given access specifier did not have to be given because an earlier access specifier of the

same type is currently active. This message is NOT given for an access specifier that is the first item to appear in a class definition. Thus

```
class abc { private: ...
```

does not draw this message. The reason this message is issued is because it is very easy to make the following mistake.

```
class A
    {
   public:   // declare private members:
     ...
   public:   // declare public members:
     ...
```

In general there are no compiler warnings that would result from such an unintentional botch.

1737 **Symbol '*Symbol*' hides global operator new** -- The indicated ***Symbol*** is a class member **operator new**. It is not compatible with the global **operator new** and, moreover, no other **operator new** within the class is argument list compatible with the global **operator new**. For this reason the user of these classes will get a surprise if he/she calls for **new X** where **X** is the class name. It will be greeted with an error. The solution is to define a single argument **operator new** as a class member. **[12, Item 9]**.

1738 **non-copy constructor '*Symbol*' used to initialize copy constructor** -- In an initializer list for a copy constructor, a base class constructor was invoked. However, this base class constructor was not itself a copy constructor. We expect that copy constructors will invoke copy constructors. Was this an oversight or was there some good reason for choosing a different kind of constructor? If this was deliberate, suppress this message. See also message **1538**.

1739 **Binary operator '*Symbol*' should be non-member function** -- The indicated function was declared as a member function. There were a number of indicators to suggest that it should have been a non-member function. The class, **X** of which it was a member has a constructor that could be used to convert numeric values to **X**. The parameter to the operator was **X** or its equivalent. For this reason the operator would behave unsymmetrically. A numeric value on the right hand side would be promoted but not a value on the left hand side. For example, **X op 27** would work but **27 op X** would not. **[12, Item 19]**.

1740 **pointer member '*Symbol*' (*Location*) not directly freed or zero'ed by destructor** -- A destructor did not free or zero a pointer member. However, it did call out to another (non-const) member function which may have done the required work. This Informational message is a companion to Warning **1540**, which covers the

situation where no member function is called from within the destructor.

1741 **member '*Symbol*' (*Location*) conceivably not initialized by constructor** -- The indicated member symbol may not have been initialized by a constructor. Was this an oversight? There appears to be a path through a loop that does initialize the member and the warning is issued because it is not clear that the loop is always executed at least once. See Section 10.1 Initialization Tracking

1742 **member '*Symbol*' (*Location*) conceivably not initialized** -- The indicated member symbol conceivably may not have been initialized by a constructor. Was this an oversight? Some of the execution paths that the constructor takes, do initialize the member. See Section 10.1 Initialization Tracking

1743 **member '*Symbol*' (*Location*) conceivably not initialized** -- The indicated member symbol conceivably may not have been initialized before use. Either this is in a constructor where it is presumed that no members are pre-initialized or this is after a statement removing its initialization such as a **delete** or a **free**. See Section 10.1 Initialization Tracking

1744 **member '*Symbol*' (*Location*) possibly not initialized by private constructor** -- The designated member was possibly not initialized by a private constructor. This message is similar to messages **1401**, **1541** and **1741**, which are given for ordinary (non private) constructors. It is given a special error number because a private constructor may be one that, by design, is never called and variables may be deliberately left uninitialized. In that case this message should be suppressed.

1745 **member '*Symbol*' (*Location*) not assigned by private assignment operator** -- The indicated Symbol was not assigned by a **private** assignment operator. This is very much like Warning **1539** except that the assignment operator in question is **private**. A **private** assignment operator may simply be a device to thwart unintended use of the assignment operator. In this case you may not care about unassigned members. If this is so, suppress this message.

The message is not given for **const** members or reference members. **[12, Item 16]**

1746 **parameter '*Symbol*' of function '*Symbol*' could be made const reference** -- The indicated parameter is a candidate to be declared as a **const** reference. For example:

```
void f( X x )
    {
    // x not modified.
    }
```

Then the function definition can be replaced with:

```
void f( const X &x )
    {
    // x not modified.
    }
```

The result is more efficient since less information needs to be placed onto the stack and a constructor need not be called.

The message is only given with class-like arguments (including struct's and union's) and only if the parameter is not subsequently modified or potentially modified by the function. The parameter is potentially modified if it is passed to a function whose corresponding parameter is a reference (not **const**) or if its address is passed to a non-const pointer. **[12, Item 22]**.

1747 **binary operator '*Symbol*' returning a reference** -- An operator-like function was found to be returning a reference. For example:

```
X &operator+ ( X &, X & );
```

This is almost always a bad idea. **[12, Item 23]**. You normally can't return a reference unless you allocate the object, but then who is going to delete it. The usual way this is declared is:

```
X operator+ ( X &, X & );
```

1748 **non-virtual base class '*Name*' included twice in class '*Name*'** -- Through indirect means, a given class was included at least twice as a base class for another class. At least one of these is not virtual. Although legal, this may be an oversight. Such base classes are usually marked virtual resulting in one rather than two separate instances of the base class. This is done for two reasons. First, it saves memory; second, references to members of such a base class will not be ambiguous.

1749 **base class '*Symbol*' of class '*Symbol*' need not be virtual** -- The designated base class is a direct base class of the second class and the derivation was specified as 'virtual'. But the base class was not doubly included (using this link) within any class in the entire project. Since a virtual link is less efficient than a normal link this may well be an unenlightened use of 'virtual'. **[23, Item 24]**. The message is inhibited if unit checkout (-**u**) is selected.

1750 **local template '*Symbol*' (*Location*) not referenced** -- A 'local' template is one that is not defined in a header file. The template was not used in the module in which it was defined.

1752 **catch parameter *Integer* is not a reference** -- This message is issued for every **catch** parameter that is not a reference and is not numeric. The problem with pointers is a problem of ownership and delete responsibilities; the problem with a

non-ref object is the problem of slicing away derivedness **[23, Item 13]**.

1753 **`Overloading special operator '`*`Symbol`*`'`** -- This message is issued whenever an attempt is made to declare one of these operators as having some user-defined meaning:

```
operator ||
operator &&
operator ,
```

The difficulty is that the working semantics of the overloaded operator is bound to be sufficiently different from the built-in operators, as to result in possible confusion on the part of the programmer. With the built-in versions of these operators, evaluation is strictly left-to-right. With the overloaded versions, this is not guaranteed. More critically, with the built-in versions of **`&&`** and **`||`**, evaluation of the 2nd argument is conditional upon the result of the first. This will never be true of the overloaded version. **[23, Item 7]**.

1754 **`Expected symbol '`*`Symbol`*`' to be declared for class '`*`Symbol`*`'`** -- The first ***`Symbol`*** is of the form: **`operator`** ***`op`*****`=`** where ***`op`*** is a binary operator. A binary operator ***`op`*** was declared for type **`X`** where **`X`** is identified by the second ***`Symbol`***. For example, the appearance of:

```
X operator+( const X &, const X & );
```

somewhere in the program would suggest that a **`+=`** version appear as a member function of class **`X`**. This is not only to fulfill reasonable expectations on the part of the programmer but also because **`operator+=`** is likely to be more efficient than **`operator+`** and because **`operator+`** can be written in terms of **`operator+=`**. **[23, Item 22]**

The message is also given for member binary operators. In all cases the message is not given unless the return value matches the first argument (this is the implicit argument in the case of a member function).

1755 **`global template '`*`Symbol`*`' (`*`Location`*`) not referenced`** -- A 'global' template is one defined in a header file. This message is given for templates defined in non-library header files. The template is not used in any of the modules comprising the program. The message is suppressed for unit checkout (**`-u`**).

1757 **`Discarded instance of post decrement/increment`** -- A postfix increment or postfix decrement operator was used in a context in which the result of the operation was discarded. For example:

```
X a;
```

```
...
a++;
```

In such contexts it is just as correct to use prefix decrement/increment. For example this could be replaced with:

```
X a;

...
++a;
```

The prefix form is (or should be) more efficient than the postfix form because, in the case of user-defined types, it should return a reference rather than a value (see **1758** and **1759**). This presumes that the side effects of the postfix form are equivalent to those of the prefix form. If this is not the case then either make them equivalent (the preferred choice) or turn this message off. **[23, Item 6]**.

1758 **Prefix increment/decrement operator '*Symbol*' returns a non-reference** -- To conform with most programming expectations, a prefix increment/decrement operator should return a reference. Returning a reference is both more flexible and more efficient **[23, Item 6]**.

The expected form is as shown below:

```
class X
    {
    X & operator++();     // prefix operator
    X operator++( int );  // postfix operator
    ...
    };
```

1759 **Postfix increment/decrement operator '*Symbol*' returns a reference.** -- To conform with most programming expectations, a postfix increment/decrement operator should return a value as opposed to a reference. **[23, Item 6]**. See example in message **1758**.

1760 **Redundant template '*Symbol*' defined identically at *Location*** -- A template was defined identically in the same module. Was this a mistake?

1761 **Declaration of function '*Symbol*' hides overloaded function '*Symbol*' (*Location*)** -- A function declaration hides an overloaded function. This does not contribute to the overloaded-ness of the function but completely hides all the overloaded functions in some prior scope. If this is your intent suppress this message.

1762 **Member function '*Symbol*' could be made const** -- The indicated (non-static) member function did not modify member data and did not call non-const

functions. Moreover, it does not make any deep modification to the class member. A modification is considered deep if it modifies information indirectly through a class member pointer. Therefore it could and probably should be declared as a **const** member function. See also Info **1763** and Elective Note **1962**.

1763 **Member function '*Symbol*' marked as const indirectly modifies class** -- The designated symbol is a member function declared as **const**. Though technically valid, the **const** may be misleading because the member function modifies (or exposes) information indirectly referenced by the object. For example:

```
class X
    {
    char *pc;
    char & get(int i) const { return pc[i]; }
    };
```

results in Info 1763 for function **X::get**. This is because the function exposes information indirectly held by the **class X**.

Experts **[24]** recommend that a pair of functions be made available in this situation:

```
class X
    {
    char *pc;
    const char & get(int i) const { return pc[i]; }
    char & get(int i) { return pc[i]; }
    };
```

In this way, if the object is **const** then only the **const** function will be called, which will return the protected reference. Related messages are also **1762** and **1962**. See also **[12, Item 29]** for a further description.

1764 **Reference parameter could be declared const reference** -- As an example:

```
int f( int & k ) { return k; }
```

can be redeclared as:

```
int f( const int & k ) { return k; }
```

Declaring a parameter a reference to **const** offers advantages that a mere reference does not. In particular, you can pass constants, temporaries and **const** types into such a parameter where otherwise you may not. In addition it can offer better documentation.

Other situations in which a **const** can be added to a declaration are covered in messages **818**, **952**, **953** and **954**.

1768 **Virtual function '*Symbol*' has an access (*String*) different from the access (*String*) in the base class (*String*)** -- An overriding virtual function has an access (public, protected or private) in the derived class different from the access of the overridden virtual function in the base class. Was this an oversight? Since calls to the overriding virtual function are usually made through the base class, making the access different is unusual (though legal).

1769 **Member or base class '*Symbol*' has no constructor** -- An initializer of the form ***Symbol*()** is ignored. ***Symbol*** is either a class member with a class, struct, or union type or is a base class. In either case the class (or struct or union) has no constructor and hence what appears to be an initialization is not. See also message **1557**.

1770 **function '*Symbol*' defined without function '*String*'** -- A typical Info 1770 message is:

```
function 'operator new(unsigned)' defined without function
    'operator delete'
```

There are three others:
operator delete without an **operator new**,
operator new[] without an **operator delete[]**, and
operator delete[] without an **operator new[]**.

In general it is not a good idea to create one of these functions without the other in the pairing. **[23, Item 27]**

You can suppress any of these without suppressing them all. Simply do a **-esym(1770,*name*)** where ***name*** is the first function named in the message.

1771 **function '*Symbol*' replaces global function** -- This message is given for **operator new** and **operator delete** (and for their **[]** cousins) when a definition for one of these functions is found. Redefining the built-in version of these functions is not considered sound programming practice. **[23, Item 27]**

1772 **Assignment operator '*Symbol*' is not returning *this** -- The assignment operator should return ***this**. This is to allow for multiple assignments as in:

```
a = b = c;
```

It is also better to return the object that has just been modified rather than the argument. **[12, Item 15]**

1773 **Attempt to cast away const (or volatile)** -- An attempt was made to cast

away **const**. This can break the integrity of the **const** system. This message will be suppressed if you use **const_cast**. Thus:

```
char *f( const char * p )
    {
    if( test() )
        return (char *) p;   // Info 1773
    else
        return const_cast<char *>(p);   // OK
    }
```

See **[12, Item 21]**.

1774 **Could use dynamic_cast to downcast ptr to polymorphic type '*Symbol*'** -- A downcast was detected of a pointer to a polymorphic type (i.e., one with virtual functions). A **dynamic_cast** could be used to cast this pointer safely. For example:

```
class B { virtual ~B(); };
class D : public B {};
...
D *f( B *p )
    {
    return dynamic_cast<D*>(p);
    }
```

In the above example, if **p** is not a pointer to a **D** then the dynamic cast will result in a NULL pointer value. In this way, the validity of the conversion can be directly tested.

B needs to be a polymorphic type in order to use **dynamic_cast**. If **B** is not polymorphic, message **1939** is issued.

1775 **catch block does not catch any declared exception** -- A catch handler does not seem to catch any exceptions. For example:

```
try { f(); }
catch( B& ) {}
catch( D& ) {}      // Info 1775
catch( ... ) {}
catch( char * ) {}  // Info 1775
```

If **f()** is declared to throw type **D**, and if **B** is a public base class of **D**, then the first catch handler will process that exception and the second handler will never be used. The fourth handler will also not be used since the third handler will catch all exceptions not caught by the first two.

If **f()** is declared to not throw an exception then Info **1775** will be issued for all four

catch handlers.

1776 **Converting a string literal to char * is not const safe (*Context*)** -- A string literal, according to Standard C++ is typed an array of **const char**. This message is issued when such a literal is assigned to a non-const pointer. For example:

```
char *p = "string";
```

will trigger this message. This pointer could then be used to modify the string literal and that could produce some very strange behavior.

Such an assignment is legal but "deprecated" by the C++ Standard. The reason for not ruling it illegal is that numerous existing functions have their arguments typed as **char *** and this would break working code.

Note that this message is only given for string literals. If an expression is typed as pointer to const **char** in some way other than via string literal, then an assignment of that pointer to a non-const pointer will receive a more severe warning.

1777 **Template recursion limit (*Integer*) reached, use -tr_limit(n) --** It is possible to write a recursive template that will contain a recursive invocation without an escape clause. For example:

```
template <class T> class A { A< A > x; };
A<int> a;
```

This will result in attempts to instantiate:

```
A<int>
A<A<int>>
A<A<A<int>>>
...
```

Using the **-vt** option (turning on template verbosity) you will see the sequence in action. Accordingly we have devised a scheme to break the recursion when an arbitrary depth of recursion has been reached (at this writing 75). This depth is reported in the message. As the message suggests, this limit can be adjusted so that it equals some other value.

When recursion is broken, a complete type is not used in the definition of the last specialization in the list but processing goes on. You can suppress this message and it will be as if nothing had happened.

1778 **Assignment of string literal to variable '*Symbol*' (*Location*) is not const safe** -- This message is issued when a string literal is assigned to a

variable whose type is a non-const pointer. The name of the variable appears in the message as well as the location at which the variable was defined (or otherwise declared). For example:

```
char *p;  p = "abc";
```

The message is issued automatically (i.e. by default) for C++. For C, to obtain the message, you need to enable the Strings-are-Const flag (**+fsc**). This message is similar to message **1776** except that it is issued whenever a string constant is being assigned to a named destination.

1780 **Returning address of reference parameter '*Symbol*'** -- The address of a parameter that has been declared as being a reference to a **const** is being returned from a function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
const int *f( const int & n )
    { return &n; }
int g();
const int *p = f( g() );
```

Here, **p** points to a temporary value whose duration is not guaranteed. If the reference is not **const** then you will get Elective Note **1940**.

This is an example of the Linton Convention as described by Murray **[21]**.

1781 **Passing address of reference parameter '*Symbol*' into caller address space** -- The address of a parameter that has been declared as being a reference to a **const** is being assigned to a place outside the function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
void f( const int & n, const int **pp )
    { *pp = &n; }
int g();
const int *p;
... f( g(), &p );
```

Here, **p** will be made to point to a temporary value whose duration is not guaranteed. If the reference is not **const** then you will get Elective Note **1940**.

This is an example of the Linton Convention as described by Murray **[21]**.

1782 **Assigning address of reference parameter '*Symbol*' to a static variable** -- The address of a parameter that has been declared as being a reference to a **const** is being assigned to a static variable. The danger of this is that the reference may

designate a temporary variable that will not persist long after the call. For example:

```
const int *p;
void f( const int & n )
    { p = &n; }
int g();
... f( g() );
```

Here, **p** will be made to point to a temporary value whose duration is not guaranteed. If the reference is not **const** then you will get Elective Note **1940**.

This is an example of the Linton Convention as described by Murray **[21]**.

1784 **Symbol '_Symbol_' previously declared as "C", compare with _Location_** -- A **_symbol_** is being redeclared in the same module. Whereas earlier it had been declared with an **extern "C"** linkage, in cited declaration no such linkage appears. E.g.

```
extern "C" void f(int);
void f(int);                // Info 1784
```

In this case the **extern "C"** prevails and hence this inconsistency probably represents a benign redeclaration. Check to determine sure which linkage is most appropriate and amend or remove the declaration in error.

1785 **Implicit conversion from Boolean (_Context_) (_Type_ to _Type_)** -- A Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of some other type. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

1786 **Implicit conversion to Boolean (_Context_) (_Type_ to _Type_)** -- A non-Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of type Boolean. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

1787 **Access declarations are deprecated in favor of using declarations** -- The C++ Standard (**[10]** section 7.3.3) specifically deprecates the use of access declarations. The preferred syntax is the using declaration. For example:

```
class D : public B
    {
    B::a;           // message 1787
    using B::a;     // preferred form and no message
    };
```

1788 **Variable '_Symbol_' (_Location_) (type '_Name_') is referenced only by**

its constructor or destructor -- A variable has not been referenced other than by the constructor which formed its initial value or by its destructor or both. The location of the symbol and also its type is given in the message. For example:

```
class A {  A(); };
void f()
    {
    A a;
    }
```

will produce a 1788 for variable '**a**' and for type '**A**'.

It very well may be that this is exactly what the programmer wants to do in which case you may suppress this message for this variable using the option **-esym(1788,a)**. It may also be that the normal use of **class A** is to employ it in this fashion. That is, to obtain the effects of construction and, possibly, destruction but have no other reference to the variable. In this case the option of choice would be **-esym(1788,A)**.

1789 **Template constructor '*Symbol*' cannot be a copy constructor** -- This message is issued for classes for which a copy constructor was not defined but a template constructor was defined. The C++ standard specifically states that a template constructor will not be used as a copy constructor. Hence a default copy constructor is created for such a class while the programmer may be deluded into thinking that the template will be employed for this purpose. **[28, Item 5]**.

1790 **Base class '*Symbol*' has no non-destructor virtual functions** -- A public base class contained no virtual functions except possibly virtual destructors. There is a school of thought that public inheritance should only be used to interject custom behavior at the event of virtual function calls. To quote from Marshall Cline, "Never inherit publicly to reuse code (in the base class); inherit publicly in order to be reused (by code that uses base objects polymorphically)" **[28, Item 22]**.

1791 **No token on this line follows the 'return' keyword** -- A line is found that ends with a return keyword and with no other tokens following. Did the programmer forget to append a semi-colon? The problem with this is that the next expression is then consumed as part of the return statement. Your return might be doing more that you thought. For example:

```
void f( int n, int m )
    {
    if( n < 0 ) return // do not print when n is negative
    print( n );
    print( m );
    }
```

Assuming **print()** returns **void**, this is entirely legal but is probably not what you intended. Instead of printing **n** and **m,** for **n** not negative you print just **m**. For **n**

negative you print **n**.

To avoid this problem always follow the return keyword with something on the same line. It could be a semi-colon, an expression or, for very large expressions, some portion of an expression.

1793 **While calling '*Symbol*': Initializing the implicit object parameter '*Type*' (a non-const reference) with a non-lvalue** -- A non-static and non-const member function was called and an **rvalue** (a temporary object) of class ***Type*** was used to initialize the implicit object parameter. This is legal (and possibly intentional) but suspicious. Consider the following.

```
struct A { void f(); };
...
A().f();          // Info 1793
...
```

In the above the 'non-static, non-const member function' is **A::f**(). The 'implicit object parameter' for the call to **A::f()** is **A()**, a temporary. Since the **A::f()** is non-const it presumably modifies **A()**. But since **A()** is a temporary, any such change is lost. It would at first blush appear to be a mistake.

The Standard normally disallows binding a non-const reference to an **rvalue** (see Error **1058**), but as a special case allows it for the binding of the implicit object parameter in member function calls. Some popular libraries take advantage of this rule in a legitimate way. For example, the GNU implementation of **std::vector<bool>::operator[]** returns a temporary object of type **std::_Bit_reference** -- a class type with a non-const member **operator=()**. **_Bit_reference** serves a dual purpose. If a value is assigned to it, it modifies the original class through its **operator=()**. If a value is extracted from it, it obtains that value from the original class through its operator **bool()**.

Probably the best policy to take with this message is to examine instances of it and if this is a library invocation or a specially designed class, then suppress the message with a **-esym()** option.

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*)**-- This kind of scenario is perhaps best explained by way of example:

```
struct A{};
// Library 1:
namespace N { int f(const A&); int f(char*); }
// Library 2:
namespace Q { int f(const A&); int f(int); }
// Non-library code:
```

```
using N::f;
using Q::f; // Info 1794 here
```

According to the ISO Standard, **[34]**, the names of **N::f(A)** and **Q::f(A)** will coexist in the global namespace (along with the names of the other overloads of **f**). This alone does not make the program ill-formed. (For that you would have to make some use of the name '**f**' that resulted in overload resolution where **f(A)** is selected.) However, a user of **N** and **Q** may be surprised to find that both library namespaces supply a function **f** that operates on **::A** objects, and that both have been introduced into the same scope. So to avoid confusion, the user may opt to do away with the using-declarations and just refer to the various **f()**'s with qualified-ids.

1795 **Defined template '*Symbol*' was not instantiated** -- The named template was defined but not instantiated. As such, the template either represents superfluous code or indicates a logic error.

The 'template' in the message could also be a temploid. A temploid is defined as either a template or a member of a temploid.

1796 **Explicit specialization of overloaded function template '*Symbol*'** --A pair of overloaded function templates was followed by an explicit specialization. For example:

```
template< class T > void f( T );
template< class T > void f( T* );
template<> void f( char * p )
    { printf( "%s\n", p ); }
```

Confusion can arise in determining which of the two function templates the specialization is actually specializing. This will lead to unexpected results when processing overload resolution since the specialization does not directly compete with the templates. Both function templates compete with each other and it can be difficult to ascertain whether the specialization is invoked in any particular call.

19.11 C++ Elective Notes

1901 **Creating a temporary of type '*Symbol*'** -- PC-lint/FlexeLint judges that a temporary needs to be created. This occurs, typically, when a conversion is required to a user object (i.e. class object). Where temporaries are created, can be an issue of some concern to programmers seeking a better understanding of how their programs are likely to behave. But compilers differ in this regard.

1902 **useless ';' follows '}' in function definition** -- It is possible to follow a function body with a useless semi-colon. This is not necessarily 'lint' to be removed but may be a preferred style of programming (as semi-colons are placed at the end of

other declarations).

1904 **`Old-style C comment`** -- For the real bridge-burner one can hunt down and remove all instances of the `/*` ... `*/` form of comment. **[12, Item 4]**

1905 **`implicit default constructor generated for class '`*`Name`*`'`** -- A default constructor was not defined for a class but a base class or a member has a non-trivial default constructor and so a non-trivial default constructor is generated for this class.

1907 **`implicit destructor generated for class '`*`Name`*`'`** -- The named class does not itself have an explicit destructor but either had a base class that has a destructor or has a member class that has a destructor (or both). In this case a destructor will be generated by the compiler. **[11 section 12.4]**

1908 **`'virtual' assumed for destructor'~`*`Name`*`()' (inherited from base class '`*`Name`*`')`** -- The destructor cited was inherited from a base class with a virtual destructor. This word '**`virtual`**' was omitted from the declaration. It is common practice to omit this keyword when implied. A warning is issued (**`1512`**) when a base class's destructor is *not* virtual. See also **`1909`**.

1909 **`'virtual' assumed, see: function '`*`Symbol`*`' (`*`Location`*`)`** -- The named function overrides a base class virtual function and so is virtual. It is common practice to omit the **`virtual`** keyword in these cases although some feel that this leads to sloppy programming. This message allows programmers to detect and make explicit which functions are actually virtual.

1911 **`Implicit call of constructor '`*`Symbol`*`' (see text)`** -- The ***`Symbol`*** in the message is the name of a constructor called to make an implicit conversion. This message can be helpful in tracking down hidden sources of inefficiencies. **[11 section 12.1]**

1912 **`Implicit call of conversion function from class '`*`Name`*`' to type '`*`Type`*`'`** -- A conversion function (one of the form ***`Symbol`*****`::operator`*****`Type`****`()`**) was implicitly called. This message can be helpful in tracking down hidden sources of inefficiencies.

1914 **`Default constructor '`*`Symbol`*`' (`*`Location`*`) not referenced`** -- A default constructor was not referenced. When a member function of a class is not referenced, you will normally receive an Informational message (**`1714`**) to that effect. When the member function is the default constructor, however, we give this Elective Note instead.

The rationale for this different treatment lay in the fact that many authors recommend defining a default constructor as a general principle. Indeed we give an Informational message (**`1712`**) when a default constructor is not defined for a class. Therefore, if you are following a modus operandi of not always defining a default constructor you may

want to turn off message **1712** and turn on message **1914** instead.

1916 **Ellipsis encountered** -- An ellipsis was encountered while processing the prototype of some function declaration. An ellipsis is a way of breaking the typing system of C or C++.

1917 **Empty prototype for *String*, assumed '(void)'** -- This message is given when an empty prototype is detected either for a function definition or for a namespace declaration (the ***String*** specifies which). Whereas we give an Informational Message (**1717**) when a (non-member) ***declaration*** contains no prototype, we give a much milder Elective Note when a ***definition*** does the same. For example:

```
int f();                // Info 1717
int f() { return 1; }   // Elective Note 1917
```

The reason for this is that the declaration form has a different meaning in C and C++. In C it is an incomplete declaration saying nothing about arguments. In C++ the declaration says there are no arguments. The definition, however, means the same in both languages. See also message **1918**. **[11 section 8.2.5]**

1918 **empty prototype for member declaration, assumed (void)** -- A function declaration within a class contains an empty prototype. This case is similar to Info **1717**, which complains about an empty prototype ***outside*** a class. It receives a lighter classification (Elective Note) because an empty prototype within a class cannot be ambiguous because C does not allow functions within classes. **[11 section 8.2.5]**

1919 **Multiple assignment operators for class '*Symbol*'** -- For a given class more than one function was declared whose name was '**operator =**'. This is not necessarily a bad thing. For example, a **String** class may very well have an assignment from **char *** and such an assignment may be advisable from an efficiency standpoint. However, it represents a loss of elegance because there will almost certainly be a **char *** constructor and an assignment operator, which will represent another way of achieving the same effect.

1920 **Casting to a reference** -- The ARM**[11]** (Section 5.4) states that reference casts are often 'misguided'. However, too many programs are openly using reference casts to place such casts in the Informational category. **[11 section 5.4]**

1921 **Symbol '*Symbol*' not checking argument against sizeof(class)** -- This note is given for either **operator new** or **operator delete** when defined as member functions. As member functions they are called when **new** (or **delete**) is applied to a class type or any derived class type. The difficulty is with the derived class type. Any specialized allocator is likely to be useless for a derived class type and hence experts suggest that a test be made of the **size_t** argument against **sizeof(class)**. Specifically PC-lint/FlexeLint is looking for one of:

```
if( arg == sizeof(class) )
if( arg != sizeof(class) )
if( sizeof(class) == arg )
if( sizeof(class) != arg )
```

or the equivalent. If any such function is found that is a member of a class that is the base of a derivation, then in addition to Note **1921**, we issue Warning **1531**. (see Steve Simpson, "More on Memory Management", Dr. Dobb's Journal, August 1994, p. 10) See also **[30, Item 51]**.

1922 **Symbol '*Symbol*' not checking argument for NULL** -- This message is given for a function **operator delete** which is not checking its parameter for being the NULL pointer. We would normally expect to see some such check as:

```
if( arg )
if( arg == 0 )
if( arg != NULL )
```

etc. Class destructors will normally filter out passing the NULL pointer into the **operator delete** so that this message is only in the Elective Note category. If there is no destructor you obtain a warning. See Warning **1532**.

1923 **macro '*Symbol*' could become const variable** -- The designated macro could probably be replaced by a **const** variable. A **const** variable is preferred in some quarters where, for example, a local debugger may not understand macros but would understand variables. **[12, Item 1]**.

The message is issued for macros containing at least one constant or constant equivalent (an earlier **const-**able macro or **const** variable) and no other variables or tokens such as ';' of a non-expression nature.

1924 **C-style cast** -- A C-style cast was detected. This can be replaced by one of the newer C++ casts having the form: ***Name*<*Type*>(*Expression*)** where ***Name*** is one of **static_cast**, **dynamic_cast**, **const_cast** or **reinterpret_cast**. **[23, Item 2]**.

1925 **Symbol '*Symbol*' is a public data member** -- The indicated ***Symbol*** is a public data member of a class. If the class is introduced with the keyword **struct** the message is not issued. In some quarters the use of public data members is deprecated. The rationale is that if function calls replace data references in the public interface, the implementation can change without affecting the interface. **[12, Item 20]**

1926 **Symbol '***Symbol***'s default constructor implicitly called** -- A member of a class (identified by ***Symbol***) did not appear in the constructor initialization list. Since it had a default constructor this constructor was implicitly called. Is this what the user intended? Some authorities suggest that all members should appear in the constructor initialization list. **[12, Item 12]**.

1927 **Symbol '*Symbol*' was not initialized in the constructor initializer list** -- A member of a class (identified by ***Symbol***) did not appear in a constructor initialization list. If the item remains uninitialized through the whole of the constructor, a Warning **1401** is issued. Some authorities suggest that all members should appear in the constructor initialization list. **[12, Item 12]**.

1928 **Symbol '*Name*' did not appear in the constructor initializer list** -- A base class (identified by ***Symbol***) did not appear in a constructor initialization list. If a constructor does not appear, then the default constructor is called. This may or may not be valid behavior. If a base class is missing from the initializer list of a copy constructor (as opposed to some ordinary constructor), then a more severe Warning (**1538**) is issued. **[12, Item 12]**.

1929 **function '*Symbol*' returning a reference** -- A non-member function was found to be returning a reference. This is not normally considered good practice because responsibility for deleting the object is not easily assigned. No warning is issued if the base class has no constructor. **[12, Item 23]**.

1930 **Conversion operator '*Symbol*' found** -- A conversion operator is a member function of the form:

```
operator Type ();
```

This will be called implicitly by the compiler whenever an object (of the class type) is to be converted to type **Type**. Some programmers consider such implicit calls to be potentially harmful leading to programming situations that are difficult to diagnose. See for example **[23, Item 5]**.

1931 **Constructor '*Symbol*' can be used for implicit conversions** -- A constructor was found that could be used for implicit conversions. For example:

```
class X
    {
   public:
    X(int);
    ...
    };
```

Here, any **int** (or type convertible to **int**) could be automatically converted to **X**. This can sometimes cause confusing behavior **[23, Item 5]**. If this is not what was intended, use the keyword '**explicit**' as in:

```
explicit X(int);
```

This will also serve to suppress this message.

1932 **Base class '*Symbol*' is not abstract.** -- An abstract class is a class with at least one pure virtual specifier. At least one author has argued **[23, Item 33]** that all base classes should be abstract although this suggestion flies in the face of existing practice.

1933 **Call to unqualified virtual function '*Symbol*' from non-static member function** -- A classical C++ gotcha is the calling of a virtual function from within a constructor or a destructor. When we discover a direct call from a constructor or destructor to a virtual function we issue Warning **1506**. But what about indirect calls. Suppose a constructor calls a function that in turn, perhaps through several levels of call, calls a virtual function. This could be difficult to detect. Dan Saks **[24]** has suggested a compromise Guideline that "imposes few, if any, practical restrictions". The Guideline, implemented by this Elective Note, issues a message whenever an unqualified virtual function is called by any other (non-static) member function (for the same '**this**' object). For example:

```
class X { virtual void f(); void g();  };

void X::g()
    {
    f();        // Note 1933
    X::f();     // ok -- non virtual call.
    }
```

Even if total abstinence is unwarranted, turning on message 1933 occasionally can be helpful in detecting situations when constructors or destructors call virtual functions.

1934 **Shift operator '*Symbol*' should be non-member function** -- It has been suggested **[12, Item 19]** that you should never make a shift operator a member function unless you're defining **ostream** or **istream** (the message is suppressed in these two cases). The reason is that there is a temptation on the part of the novice to, for example, define output to **ostream** as a class member function left shift that takes **ostream** as an argument. This is exactly backwards. The shift operator normally employs the destination (or source) on the left.

On the other hand, if the class you are defining is the source or destination then defining the shift operators is entirely appropriate.

1935 **Dynamic initialization for class object '*Symbol1*' (references '*Symbol2*')** -- A static class-like object whose name is ***Symbol1*** is dynamically initialized by referencing ***Symbol2*** (the latter is normally a constructor for the former). The reason for noting this initialization is that the order of inter-module dynamic initializations is not defined. (Within a module, however, the intializations are done in the order of appearance.) Hence, if the constructor is itself dependent on dynamic initialization occurring in another module the behavior is undefined. For example:

```
class X
    {  X(): ... };

X x:
```

This will elicit Elective Note **1935** that **x** is being initialized dynamically by a call to **X::X()**. Now, if this constructor were to be accessing information that depended on the order of evaluation (such as accessing the value of **x** itself) the result would be undefined. We have no evidence of this at this point. and for this reason the message is in the Elective Note category. However, programmers with a suspected order-of-initialization problem will probably want to turn this on. See also **1936**, **1937**, **1544** and **1545**.

1936 **Dynamic initialization for variable '*Symbol1*' (references '*Symbol2*')** -- A static scalar whose name is ***Symbol1*** is dynamically initialized and references ***Symbol2*** during the initialization. For example, let a module consist only of:

```
int f();
int n = f();
```

Here we report that **n** is dynamically initialized by **f()**. There may be other symbols referenced, ***Symbol2*** is just the first. The reason for noting this initialization is that the order of inter-module dynamic initializations is not defined. (Within a module, however, the initializations are done in the order of appearance.) If ***Symbol2*** were a variable, then PC-lint/FlexeLint could determine that the variable is dynamically initialized in another module and issue a **1544** or **1545** as appropriate. However, the symbol referenced could be a function (as in the example) and PC-lint/FlexeLint does not analyze the complete function call graph to determine whether there is a dependency on another dynamic initialization. See also **1935** and **1937**.

1937 **Static variable '*Symbol*' has a destructor.** -- A static scalar whose name is ***Symbol*** has a destructor. Destructors of static objects are invoked in a predictable order only for objects within the same module (the reverse order of construction). For objects in different modules this order is indeterminate. Hence, if the correct operation of a destructor depends on the existence of an object in some other module an indeterminacy could result. See also **1935**, **1936**, **1544** and **1545**.

1938 **constructor '*Symbol*' accesses global data.** -- A constructor is accessing global data. It is generally not a good idea for constructors to access global data because order of initialization dependencies can be created. If the global data is itself initialized in another module and if the constructor is accessed during initialization, a 'race' condition is established. **[12, Item 47]**

1939 **Down cast detected** -- A down cast is a cast from a pointer (or reference) to a base class to a pointer (or reference) to a derived class. A cast down the class hierarchy is

fraught with danger. Are you sure that the alleged base class pointer really points to an object in the derived class? Some amount of down casting is necessary, but a wise programmer will reduce this to a minimum. **[12, Item 39]**

1940 **Address of reference parameter '*Symbol*' transferred outside of function** -- The address of a reference parameter is being transferred (either via a return statement, assigned to a static, or assigned through a pointer parameter) to a point where it can persist beyond the lifetime of the function. These are all violations of the Linton Convention (see Murray **[21]**).

The particular instance at hand is with a reference to a non-const and, as such, it is not considered as dangerous as with a reference to a **const**. (See **1780**, **1781** and **1782** for those cases). For example:

```
int *f( int &n ) { return &n; }
int g();
int *p = f( g() );
```

would create a problem were it not for the fact that this is diagnosed as a non-**lvalue** being assigned to a reference to non-const.

1941 **Assignment operator for class '*Symbol*' does not return a const reference to class** -- The typical use of an assignment operator for class **C** is to assign new information to variables of class **C**. If this were the entire story there would be no need for the assignment operator to return anything. However, it is conventional to support chains of assignment as in:

```
C x, y, z;
...
x = y = z;
// parsed as x = (y = z);
```

For this reason assignment normally returns a reference to the object assigned the value. For example, assignment **(y = z)** would return a reference to **y**.

Since it is almost never the case that this variable is to be reassigned, i.e. we almost never wish to write:

```
(x = y) = z;    // unusual
```

as a general rule it is better to make the assignment operator return a **const** reference. This will generate a warning when the unusual case is attempted.

But experts differ. Some maintain that in order to support non-const member functions operating directly on the result of an assignment as in:

```
(x = y).mangle();
```

where, as its name suggests, **mangle** is non-const it would be necessary for the return value of assignment to be non-const. Another reason to not insist on the **const** qualifier is that the default assignment operator returns simply a reference to object and not a reference to **const** object. In an age of generic programming, compatibility may be more important than the additional protection that the **const** would offer.

1942 **Unqualified name *'Symbol'* subject to misinterpretation owing to dependent base class** -- An unqualified name used within a class template was not found within the calls and, moreover, the class has at least one dependent base class. There is a potential ambiguity here. According to the standard, the dependent base class should not be searched either at template definition time or at template instantiation time. Nonetheless, some implementations do make that search at instantiation time. Even if the compiler is conforming, the implementator or even a reader of the code may think the base class is searched leading to confusion.

To eliminate the ambiguity, the name should be full qualified (or referenced using **this**). For example:

```
class X;
template< class T >
     class A : T
          {
          X *p;
          bool f() { return y; }
          }
```

Both the reference to **X** (on line 5) and to **y** (on line 6) will be flagged. One possible modification is:

```
class X;
template< class T >
     class A : T
          {
          ::X *p
          bool f() { return this->y; }
          };
```

This solidifies and disambiguates the code. The reference to **X** is guaranteed to be the **X** on line 1 and can no longer be high-jacked by the base class. Also, since **y** is not a member of **A**, the class will not instantiate unless **y** is found to be a member of the base class.

1960 **Violates MISRA C++ Required Rule *Name*, *String*** -- In addition to a C coding standard, MISRA has also compiled a C++ standard. The list of checks made are as follows:

(Rule 0-1-8) Void return type for function without external side-effects.
(Rule 2-13-2) Octal constant or escape sequence used.
(Rule 2-13-4) Lower `case` literal suffix.
(Rule 3-1-1) Object/function definitions in headers.
(Rule 3-1-2) Function not declared at file scope.
(Rule 5-0-3) Implicit conversion of `cvalue`.
(Rule 5-0-4) Implicit conversion changes signedness.
(Rule 5-0-5) Implicit conversion between integer and floating point types.
(Rule 5-0-6) Implicit conversion to smaller type.
(Rule 5-0-7) Cast `cvalue` between integer and floating point types.
(Rule 5-0-8) Cast of `cvalue` to larger type.
(Rule 5-0-9) Cast of `cvalue` changes signedness.
(Rule 5-0-10) Recasting required for operator '`~`' and '`<<` '.
(Rule 5-0-19) More than two pointer indirection levels used.
(Rule 5-0-21) Bitwise operator applied to signed underlying type.
(Rule 5-2-12) Array-pointer decay when passing the array to a function
(Rule 5-3-2) Prohibited operator applied to unsigned underlying type.
(Rule 5-3-3) Overloading unary `&`.
(Rule 5-3-4) '`sizeof`' used on expressions with side effects.
(Rule 5-14-1) Side effects on right hand side of logical operator.
(Rule 5-18-1) Comma operator used.
(Rule 6-2-3) Null statement not in line by itself.
(Rules 6-3-1 Left brace expected for `for`, `do`, `switch` and `while`.
(Rule 6-4-1) Left brace expected for `if`, `else`.
(Rule 6-4-2) No `else` at end of '`if ... else if` ' chain.
(Rule 6-4-7) Boolean value in switch expression.
(Rule 6-6-2) Gotos jumping to an earlier point in the code.
(Rule 6-6-3) `continue` statement should not be used.
(Rule 6-6-4) More than one `break` terminates loop.
(Rule 7-3-1) Global declarations other than `main()`, namespace declarations, `extern "C"` declarations and arithmetic typedefs.
(Rule 7-3-2) Using the identifier main for fuctions other than global one.
(Rule 7-3-3) Unnamed namespaces in headers.
(Rule 7-3-4) No using-directives allowed.
(Rule 7-3-6) using-directives or using declarations (except class and/or block scope using declaration) in header files.
(Rule 8-0-1) Multiple declarators in a declaration.
(Rule 8-4-1) Function has variable number of arguments.
(Rule 8-5-3) Should initialize either all enum members or only the first.
(Rule 9-5-1) Unions shall not be used.
(Rule 14-8-1) Explicit specialization of overload function templates.
(Rule 15-1-3) Empty throw outside of a catch block.
(Rule 15-3-7) Catch handler after `catch(...)` in a try-catch sequence.
(Rule 16-0-1) Only preprocessor statements and comments before '`#include`'.
(Rule 16-0-2) '`#define/#undef`' used within a block.

(Rule 16-0-3) Use of '**undef**' is discouraged.
(Rule 16-0-4) Use of function-like macros is discouraged.
(Rule 16-1-1) Non-standard use of '**defined**' preprocessor operator.
(Rule 16-2-4) Header file name with non-standard character.
(Rule 16-3-1) Multiple use of '#' and/or '##' operators in macro definition.

You may disable individual rules to your taste by using the Rule number in an esym option, just as with message **960**.

See **[37]** for information on the MISRA guidelines. See **Section 13.12 MISRA Standards Checking** for information on activating Lint's MISRA checking in general.

1961 **virtual member function '*Symbol*' could be made const** -- This message is similar to message **1762** (member function could be made const) except that it is issued for a virtual function. You may not want to make virtual functions const because then any overriding function would have to be const as well. Consider, for example:

```
class A { virtual void f() {} /* ... */ };
class B : public A
    { int a; void f() { a = 0; } };
```

Here, class **B** overrides **A**'s function **f()** and, in doing so, modifies member **a**. If **A::f()** had been declared const, this would not have been possible.

Nonetheless, a particularly rigorous user may want to hunt down such virtual functions and make them all const and so this Note is provided.

This message is also similar to Note **1962**, which is issued for functions that make deep modifications. Note **1962** takes priority over **1961**. That is, a virtual function that makes a deep modification (but no shallow modifications) will have Note **1962** issued but not Note **1961**.

1962 **Non-const member function '*Symbol*' contains a deep modification.** -- The designated member function could be declared **const** but shouldn't be because it contains a deep modification. For example:

```
class X
    {
    char *p;
    public:
    void f() { *p = 0; }
    x();
    };
```

will elicit this message indicating that **X::f()** contains a deep modification. A modification is considered shallow if it modifies (or exposes for modification) a class member directly. A modification is considered deep if it modifies information indirectly

through a class member pointer. This Elective Note is available for completeness so that a programmer can find all functions that could result in a class being modified. It does not indicate that the programming is deficient. In particular, if the function is marked **const** an Info **1763** will be issued. See also **1762**, **1763**.

1963 **Violates MISRA C++ Required Rule *Name*, *String*** -- This message is issued for some violations of the MISRA C++ advisory guidelines. The list of checks made are as follows:

(Rule 2-5-1) Possible digraph used.
(Rule 5-0-2) Dependence placed on C's operator precedence.
(Rule 14-8-2) Mixing template and non-template functions in a viable set.
(Rule 15-0-2) Throwing a pointer expression.
(Rule 16-2-5) Header file name with non-standard character.
(Rule 16-3-2) No use of '#' or '##'.

This message can be suppressed based on rule number. See also Message **960**.

See **[37]** for information on the MISRA guidelines. See **Section 13.12 MISRA Standards Checking** for information on activating Lint's MISRA checking in general.

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 MSRA-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 "**<**", "**>**" and "**&**" 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*

21. BIBLIOGRAPHY

[1] Kernighan, Brian and Dennis Ritchie,
The C Programming Language,
Prentice Hall, Englewood Cliffs,
1978 (First Edition), 1988 (Second Edition).

[2] *ISO/IEC 9899:1990*
International Standard-
Programming languages – C.

[3] Harbison, S.P. and G.L. Steele, Jr.,
C: A Reference Manual,
Prentice Hall, Englewood Cliffs NJ,
1984 (First Edition), 1988 (Second Edition), 1995 (4th Edition)
ISBN 0-13-326232-4.

[4] ISO/IEC 9899:1999
Programming languages - C
//www.dkuug.dk/JTCI/SC22/WG14.
Available in Adobe PDF format for $18 from
//www.techstreet.com/ncitsgate.html

[5] Ward, Robert,
Debugging C,
Que Corporation, Indianapolis IN, 1986.

[6] Jaeschke, Rex,
Portability and the C Language,
Hayden Books, Indianapolis IN, 1989.

[7] Hatton, Les,
Safer C,
McGraw-Hill, London/New York, 1995.

[8] Van Der Linden, Peter
Expert C Programming - Deep C Secrets,
Prentice Hall, Englewood Cliffs NJ, 1994.

[9] *Guidelines for the Use of the C Language in Vehicle Based Software (MISRA)*
The Motor Industry Research Association, April 1998
Warwickshire, UK, //www.misra.org.uk

[10] *ISO/IEC 14882:1998*
International Standard – Programming Languages – C++
Available in Adobe PDF format for $18 from
//www.techstreet.com/ncitsgate.html

[11] Ellis, M. A. and Stroustrup, B.
The Annotated C++ Reference Manual,
Addison-Wesley, Reading, MA,
First Printing 1990, Reprinting w/corrections, May 1992.

[12] Meyers, Scott
Effective C++
Addison-Wesley, Reading MA, 1992

[13] Cargill, Tom
C++ Programming Style
Addison-Wesley, Reading, MA, 1992

[14] Coplien, James
Advanced C++ Programming Styles and Idioms
Addison-Wesley, Reading, MA, 1991

[15] Eckel, Bruce
C++ Inside and Out
Osborne / McGraw-Hill, 1992

[16] Hekmatpour, S.
C++: A Guide for Programmers
Prentice Hall, 1992

[17] Plum, T. and Saks, D.
C++ Programming Guidelines
Plum Hall, 1991

[18] Stroustrup, Bjarne
The C++ Programming Language., 2nd Ed.
Addison-Wesley, Reading, MA, 1992

[19] Koenig, Andrew
Check List for Class Authors
The C++ Journal, 2:1 (1992 Nov 1), 42-46
Reprinted in "Ruminations on C++" **[26]**, Chapter 4

[20] Cargill, Tom
C++ Gotchas
presented at C++ World, November 1992.

[21] Murray, Robert B.
C++ Strategies and Tactics
Addison-Wesley, Reading MA, 1993 ISBN 0-201-56382-7

[22] Spuler, David A.
C++ and C Debugging, Testing and Reliability
Prentice Hall, Englewood Cliffs, NJ, 1994.

[23] Meyers, Scott,
More Effective C++,
Addison-Wesley, Reading MA, 1996

[24] Saks, Dan,
C++ Gotchas!,
Saks & Associates, 393 Leander Dr., Springfield, OH.

[25] Reznick, Larry
Tools for Code Management,
R&D Books, Lawrence, KS 1996

[26] Koenig, Andrew and Barbara Moo
Ruminations on C++,
Addison-Wesley, Reading, MA ISBN 0-201-42339-1

[27] Holub, Allen I.
Enough Rope to Shoot Yourself in the Foot,
McGraw Hill, 1995.

[28] Sutter, Herb,
Exceptional C++,
Addison-Wesley, Reading MA, 2000, ISBN 0-201-61562-2

[29] Sutter, Herb and Andrei Alexandrescu,
C++ Coding Standards (101 Rules, Guidelines, and Best Practices),
Addison-Wesley, Reading MA,2005, ISBN 0-321-11358-6

[30] Meyers, Scott,
Effective C++ Third Edition,
Addison-Wesley, Reading MA, 2005, ISBN 0-321-33487-6

[31] Lewis, Bil and Daniel J. Berg,
Multithreaded Programming with Pthreads,
Sun Microsystems Press, 1998, ISBN 0-13-680729-1

[32] Vandevoorde, David and Nicolai M. Josuttis,
C++ Templates -- The Complete Guide,
Addison-Wesley, Boston, 2003, ISBN 0201734842

[33] The Motor Industry Software Reliability Association
MISRA-C:2004 Guidelines for the use of the C Language in critical systems,
The Motor Industry Research Association,
Warwickshire, UK 2004, ISBN 0952415690

[34] ISO/IEC 14882:2003
International Standard -- Programming Languages -- C++

[35] ISO/IEC 14882
C++ Standard Core Language Defect Reports
//open-std.org/jtcl/sc22/wg21/docs/cwg_defects.html

[36] Saks, Dan,
const T vs. T const,
www.dansaks.com, Published Articles, 1999.

[37] The Motor Industry Software Reliability Association
MISRA-C++: 2008 Guidelines for the use of the C++ Language in critical systems,
The Motor Industry Research Association,
Warwickshire, UK 2008.

Appendix A

Option Summary

----- Error Inhibition Options -----

(**-** inhibits and **+** enables error messages)

(**#** and ***Symbol*** may include wild-card characters '**?**' and '*****')

-e# Inhibit message number **#**
-e(#) Inhibit for next expression
-e{#} Inhibit for next **{region}**
-eai Args differ sub-integer
-eas Args same size
-ecall(#,*Func*) By number, fnc call
-efunc(#,*Func*) By number, function
+efreeze(*wlvl*) Freeze for ***lvl***
-elib(#) Within library headers
-elibmacro(#) For all library macros
-emacro(#,*Symbol*) Within macro
--emacro((#),*Symbol*) Within expr macro
--emacro({#},*Symbol*) Within stmt macro
-epnc Pointers to nominal chars
-eps Pointers to same size
-epuc Pointers to sgnd-unsgnd chars
-esym(#,*Symbol*) By number, symbol
-etd(*TypeDiff*) Ignore type diff.
-etype(#,*TypeName*) By number, type
++limit(*n*) Locks in limit of ***n***
-restore Resets error inhibitions
-*wlvl* Set warning level (0,1,2,3,4)
-zero Sets exit code to 0

!e# Inhibit message **#** this line
--e(#) For entire current expression
--e{#} For entire current **{region}**
-ean Args differ nominally
-eau Args differ signed-unsigned
-efile(#,*File*) By number, file
+efreeze Disable Message inhibition
++efreeze[(*wlvl*)] Deep-freeze ***lvl***
-elibcall(#) Calls to library functions
-elibsym(#) For all library symbols
-emacro((#),*Symbol*) Within expr macro
-emacro({#},*Symbol*) Next stmt macro
-epn Pointers to nominal
-epp Pointers are pointers
-epu Pointers to signed-unsigned
-estring(#,*String*) By number, string
+esym(#,*Symbol*) Enable by no. symbol
-etemplate(#) In template expansion
-limit(*n*) Limits number of messages
-save Saves error inhibitions
-restore_at_end Restores at module end
-wlib(*lvl*) Library warning level
-zero(#) Like **-zero** unless msg no. **#**

------ Verbosity Options ------

Format: **-/+v[aceh-iostw#]{mf*int**}**

-v... Output to stdout only
-v Turn off verbosity (note absence of option letters)
+v... Output to stderr and also to stdout
+v Don't change options but output to stderr and stdout

Zero or more of:

a Attempts to open

c Unique Calls	**e** Function templates
h Dump strong type hierarchy	**h-** Compressed form of **h**
i Indirect files	**o** Display options
s Storage consumed	**t** Template expansions
w Specific Walks	**#** Append file ID nos.

One of:

m Module names (the default)	**f** Header Files (implies **m**)
int Every ***int*** lines (implies **f**)	***** All verbosity

----- Message Presentation Options -----

-h[**abefFrsSm/<M>/<I>**]**<ht>** message height (default = **-ha_3**)

a Position indicator Above line	**b** Indicator Below line
f Frequent file information	**F** Always produce file info
e Place source line @ End of msg	**r** Repeat source line each msg
s Space after each non-wrapup msg	**S** Space after each msg
m/*M*/ Macro indicator is ***M***	**mn** Turn off macro indication
I The position Indicator	***ht*** Height of messages

-width(*Width*,*Indent*) Message width (default = **-width(79,4)**)
-append(*errno*,*msg*) Appends ***msg*** for message ***errno***

-format=... Specifies message format
-format4a=, **-format4b=** Specifies format if msg. **ht. = 4**
-format_specific=... Prologue to specific Walk messages
-format_stack=... Format for output of **-stack**
-format_template=... Format for prologue to template instantiation
-format_verbosity=... Format for verbosity under **+html**

format codes (%...)

%c column no.	**%l** Line no.	**\t** Tab
%C Column no. + 1	**%m** Msg text	**\s** Space
%f Filename	**%n** msg Number	**\a** Alarm
%i functIon name	**%t** msg Type	**\q** Quote
%(...**%)** Use ... if '**%f**' or '**%i**' are non null		**** Backslash
		\n Newline

format escapes (\...)

+source(*suboptions*) Echos entire source files(s); suboptions:

-number	Do not number lines	**-indent**	Do not indent hdr lines
-m(*files*)	Ignore given modules	**+h(*hdrs*)**	Echo given ***hdrs***
-h(*hdrs*)	Do not echo ***hdrs***	**+dir(*dirs*)**	Echo hdrs from ***dirs***
-dir(*dirs*)	Do not echo hdrs from ***dirs***	**+class(all)**	Echo all hdrs
+class(*project*)	Echo project hdrs		

+html(*options*) Output in html format (example in **env-html.lnt**)
version(...) Can be used to specify the version of html
head(*file*) Includes ***file*** just after ***html***
+xml([***name***]**)** Activate escapes for xml (example in **env-xml.lnt**)

If name is provided, output appears within ***name*** ... ***/name***

-message(*String*) Output ***String*** as an informational message

-summary([***out-file***]**)** Issues or appends a summary of error messages

-t*#* Sets tab size to *#*

+typename(*#*) Includes types of Symbols in message *#*

+xml(*name*) Format output messages in **xml**

.... message presentation flags ....

ffn use Full file Names (OFF)
flm Lock Message format (OFF)
fsn treat Strings as Names (OFF)
ffo Flush Output each msg (ON)
frl Reference Location info (ON)

_____ DATA GROUP _____

--- Scalar Data Size and Alignment Options (default value(s)) ---

-sb#	bits in a byte (8)	-sbo#	**sizeof(bool)** (1)
-sc#	**sizeof(char)** (1)	-slc#	**sizeof(long char)** (2)
-ss#	**sizeof(short)** (2)	-si#	**sizeof(int)** (4)
-sl#	**sizeof(long)** (4)	-sll#	**sizeof(long long)** (8)
-sf#	**sizeof(float)** (4)	-sd#	**sizeof(double)** (8)
-sld#	**sizeof(long double)** (16)	-smp#	size of all member ptrs (4)
-smpD#	size of mem ptr (data) (4)	-smpFP#	size, mem ptr (Far Prog) (4)
-smpNP#	size, mem ptr (Near Prog) (4)	-smpP#	size of mem ptrs (prog) (4)
-sp#	size of(all pointers) (4 6)	-spD#	size of both data ptrs (4 6)
-spF#	size of both far ptrs (6)	-spFD#	size of **far** data pointer (6)
-spFP#	size of **far** prog pointer (6)	-spN#	size of both **near** ptrs (4)
-spND#	size of **near** data pointer (4)	-spNP#	size of **near** prog pointer (4)
-spP#	size of both program ptrs (4 6)	-sw#	size of wide **char** (2)

-a*code#* Specifies alignment, ***code*** is any code used above in **-s*code#***
= 1, no alignment; *#* = 2, 2-byte boundary; etc. By default, a type's alignment is the largest power of 2 that divides the size of the type

-align_max(*n*) Set the maximum alignment to ***n***
-align_max(push) Saves the current maximum alignment
-align_max(pop) Restores previously pushed maximum alignment

.... scalar data flags ....

fba Bit Addressability (OFF)
fbc Boolean Constants 0b... (OFF)

fbo Activate bool, true, false (ON)
fcu char-is-unsigned (OFF)
fdc (C++) Distinguish plain **char** (ON)
fdl pointer-Diff.-is-long (OFF)
fis Integral consts. are Signed (OFF)
flc allow **long char** (OFF)
fll allow **long long int** (OFF)
fmd Multiple Definitions (OFF)
fpd Pointers Differ in size (OFF)
fsc Strings are **const char *** (OFF)
fsu String **unsigned** (OFF)
fwc use internal **wchar_t** (OFF)
fwm Microsoft **wprintf** formatting (OFF)
fwu wchar_t is unsigned (OFF)

------- Data Modifier Options -------

-mS Small model
-mD large Data model
-mP large Program model
-mL Large program and data

.... data modifier flags ....

f@m **@** is a Modifier (OFF)
fat Parse **.net** ATtributes (ON)
fcd cdecl is significant (OFF)
fem allow Early Modifiers (OFF)
fiq Ignore default Qualifier (OFF)
fqb Qualifiers Before types (ON)

------ struct, union, class, enum, namespace Flags ------

fab ABbreviated structures (OFF)
fan ANonymous unions (OFF)
fas Anonymous Structures (OFF)
fbu force Bit fields Unsigned (OFF)
fct Create Tags (OFF)
feb Enum's can be Bitfields (ON)
fie Integer-model-for-Enums (OFF)
fld Label Designator (OFF)
fns Nested Struct (ON)
fnt (C++) Nested Tags (ON)
fsg Std is Global (OFF)
fss regard Sub-Struct as base (ON)
fus Using namespace std (OFF)
fvl Variable Length arrays (OFF)

_____ PROCESSING GROUP _____

------- Preprocessor Options -------

-d*name*[**=***value*]	Defines preprocessor symbol
-d*name***{***definition***}**	For use with **-scavenge**
-D*nm*[**=***val*][**;***nm*[**=***val*]]...	Define set of symbols
+d... or **+D**...	Same as **-d** or **-D** except it locks in a definition
-d*name***()**[**=***value*]	Define function-like macro
-#d*name*[**=***value*]	Defines symbol (for **#include** only)
-header(*file***)**	Auto-includes file in each module
-i*directory*	Search directory for **#include**
-incvar(*name***)**	Change name of INCLUDE environment variable

-/+macros	Halve/double the maximum macro size
-pch(*hdr*)	Designates ***hdr*** as the pre-compiled header
-ppw(*word*[,...])	Disables(**-**) or enables(**+**) preprocessor words
+ppw(*word*[,...])	eg: **+ppw(ident)** enables **#ident**
--ppw(*word*[,...])	Removes built-in meaning of word
-ppw_asgn(*w1*,*w2*)	Assigns pre-proc meaning of **w2** to **w1**
+pragma(*name*,*action*)	Associates ***action*** with ***name***; ***action*** is one of
off	Turns processing off (as with assembly code)
on	Turns processing back on
once	Physically include this file just once
message	Issue a message
macro	**pragma_*name*** becomes a macro
fmacro	**pragma_*name*** becomes a function-like macro
options	**pragma_*name*_suboption** becomes a function macro
ppw	The pragma becomes a preprocessor command
push_macro	**push_macro("nm")** saves the current definition of **nm**
pop_macro	**pop_macro("nm")** restores a pushed definition
-pragma(*name*)	Disables pragma ***name***
-u*name*	Undefines ***name***
--u*name*	Ignore past and future defines of ***name***

.... preprocessor flags ....

fce Continue-on-Error (OFF)
fep Extended Preprocessor exps. (OFF)
fim -i can have Multiple dirs. (ON)
fln activate **#line** (ON)
fps Parameters within Strings (OFF)

--------- Tokenizing Options ---------
see also Compiler Adaptation Keywords

-$	Permits **$** in identifiers
-ident(*chars*)	Add identifier characters
-ident1(*char*)	Define a 1-**char** identifier
+linebuf	Doubles size of line buffer
-rw(*word*[,...])	Disables(**-**) or enables(**+**) reserved words ...
+rw(*word*[,...])	***word* = '*ms'** implies all MS keywords
--rw(*word*[,...])	Removes built-in meaning of ***word***
-rw_asgn(*w1*,*w2*)	Assigns reserved word meaning of **w2** to **w1**

.... flags affecting tokenization ....

fnc Nested Comments (OFF)
ftg permit Tri Graphs (ON)

-------- Parsing Options ---------

-fallthrough	A switch case allowing flow from above
-unreachable	A point in a program is unreachable

.... parsing flags ....

ffb for clause creates Block (ON)
flf (C++) process Lib Func defs (OFF)
fna (C++) allow '**operator new[]**' (ON)
fpc Pointer Casts retain lvalue (OFF)
fpm Precision is Max of args (OFF)

-------- Template Options --------

-tr_limit(*n*)	Sets a template recursion limit
-template(X)	Hex constant **X** sets esoteric template flags

.... template flags ....

ftf raw Template Functions (OFF)

---- Compiler-adaptation Options ----

-A	Specifies strict ANSI/ISO
-A(C*year*)	Specifies the year of the assumed C standard
-A(C++*year*)	Specifies the year of the assumed C++ standard
-c*code*	Identifies the compiler
-a#*predicate*(tokens)	Assert the truth of **#*predicate*** for tokens (Unix)
+/-compiler(*flag*[,...])	Sets/resets ***flag***; default value shown by (ON/OFF)
std_digraphs	(OFF) Enables '**<:**' and '**:>**' with standard meaning
base_op	(OFF) Enables digraph '**:>**' with ancient meaning
std_alt_keywords	(OFF) Enables keywords: **and**, **and_eq**, **bitand**, **bitor**, **compl**, **not**, **not_eq**, **or**, **or_eq**, **xor**, **xor_eq**
-overload(X)	Hex constant **X** sets esoteric overload resolution flags
-plus(*Char*)	Identifies ***Char*** as an alternate option character equiv to **+**
-scavenge(*filename-pattern*[,...])	Turns lint into a scavenger of macro names within files matching ***filename-pattern***
-scavenge(clean,*file*)	Used subsequently to clean up file which bears the results of compiler macro replacement
-template(X)	Hex constant **X** sets esoteric template flags

.... compiler-adaptation reserved words (keywords) ....

`@`	Ignore expression to the right
`__assert`	'Ideal' ***assert*** function
`___assert`	Like `_assert` but it always returns
`_bit`	1 bit wide type
`_gobble`	Ignore next token
`_ignore_init`	Ignore initializer for data and ... ignore function body for functions
`_to_brackets`	Ignore next parenthesized (or bracketed) expression
`_to_semi`	Ignore until `;`
`_to_eol`	Ignore until end-of-line
`_up_to_brackets`	Ignore up to and including a bracketed expression
`__packed`	struct data members are packed
`__typeof__(e)`	Like `sizeof` but returns the type of ***e***

---------- Old C Flags ----------

`fdr` Deduce-Return-mode (OFF)
`fkp` K&R Preprocessor (OFF)
`fsa` Structure-Assignment (ON)
`fva` Variable Arguments (OFF)
`fvr` Varying-Return-mode (OFF)
`fxc` eXact `char` arg. (OFF)
`fxs` eXact short arg. (OFF)
`fzu` siZeof-is-Unsigned (ON)
`ffd` promote Floats to Double (OFF)
`fmc` Macro Concatenation (OFF)
`ful` Unsigned `long` (ON)
`fvo` VOid data type (ON)
`fxa` eXact Array arg. (OFF)
`fxf` eXact Float arg. (OFF)
`fzl` siZeof-is-Long (OFF)

_____ SPECIAL DETECTION GROUP _____

------- Strong Type Options -------

`-strong(`***`Flags,Type(s)`***`)` Check strong types ..., Flags are:
- `A[irpacz]` == on Asgn to (except Init, Ret, Param, Asgn op, Consts, Zero);
- `J[erocz]` == on Joining (except Eqlty, Rel, Other ops, Constants, Zero);
- `X` == on eXtraction;
- `l` == allow library;
- `B`[`f`] or `b`[`f`] == strong and weak Boolean (`f` == length-1 bit fields are NOT Boolean)

`-index(`***`flags,ixtype,type(s)`***`)` Establish ***`ixtype`*** as index type
flags: `c` == allow Constants, `d` == allow Dimensions

`-parent(`***`Parent,Children`***`)` Augment strong type hierarchy

`-father(`***`Parent,Children`***`)` A stricter version of `parent`

.... strong type flags ....

fhd Hierarchy Down warning (ON)
fhg Hierarchy uses Graph. chars (ON)
fhs Hierarchy of Strong types (ON)
fhx Hierarchy of indeX types (ON)

-------- Semantic Options --------

-function(*f0*,*f1*, ...)	Assign semantics of ***f0*** to ***f1***, ...
-printf(*#*,*f1*, ...)	***f1***, ... are **printf**-like, *#* is arg. no. of format
-scanf(*#*,*f1*, ...)	***f1***, ... are **scanf**-like, *#* is arg. no. of format
-printf_code(*Code*[,*Type*])	Allows user-defined **printf** codes
-scanf_code(*Code*[,*Type*])	Allows user-defined **scanf** codes
-sem(*fnc*,*sem1*, ...)	Associates a set of semantics with a function
-wprintf(*#*,*f1*, ...)	Wide char version of **-printf**
-wscanf(*#*,*f1*, ...)	Wide char version of **-scanf**

.... semantics (i.e. arguments to the -sem option) ....

4 kinds of semantics: **0** == overall, **a** == argument, **r** == return, **f** == flag

kind:	semantic:	meaning:
r	**r_null**	function may return NULL
r	**r_no**	function does not return
f	**initializer**	function initializes all data members
f	**cleanup**	function clears all pointer members
a	**#p**	#th argument must not be NULL
a	**custodial(#)**	#th argument takes custody
a	**type(#)**	#th argument is reflected into the return type
a	**pod(#)**	#th argument must be POD
a	**nulterm(#)**	#th argument is nul-terminated
a	**inout(#)**	#th argument is read and written
f	**pure**	the function is a pure function
r	expression in the form of a predicate containing an @ operator is assumed true and return value deduced	
0	any other expression is considered a function requirement	

.... expression components: ....

#n = integer value of #th arg
#p = item count of #th arg
#P = byte count of #th arg
integer = itself
name of **macro**, **enum**, **const** = current value of same
malloc(*exp*) = **malloc**'ed area of size ***exp***
new(*exp*) = **new**'ed area of size ***exp***
new[](*exp*) = **new[]**ed area of size ***exp***

() Unary operators: **+ - | ~**
Binary operators: **+ - * / % < <= == != > >= | & ^ << >> || &&**
Ternary operator: **?:**

.... thread semantics ....

thread = the function is a thread
thread_mono = a single-instance thread
no_thread = function is not a thread
thread_create(#) = #th argument is a thread
thread_lock = function locks a mutex
thread_unlock = function unlocks a mutex
thread_unsafe = unsafe to call by multiple threads
thread_unsafe(*groupid*) = group can't be called by multiple threads
thread_safe = not **thread_unsafe**
thread_protected = function is protected by critical section
thread_not(*list*) = lists threads that may not invoke function
thread_only(*list*) = lists the only threads that may invoke function

.... semantic flags ....

ffc Function takes Custody (ON)

----- Thread Options -----
(see also thread semantics)

prefix '**+**' designates a property and '**-**' reverses it

+/-thread_unsafe_h(*hdr*[,...]) ftns in headers are **thread_unsafe**
+/-thread_unsafe_group_h(*hdr*[,...]) ftns in hdrs form a **thread_unsafe** group
+/-thread_unsafe_dir(*dir*[,...]) hdrs in dirs house **thread_unsafe** functions
+/-thread_unsafe_group_dir(*dir*[,...]) hdrs in dirs form a **thread_unsafe** group

----- Value Tracking Options -----

-passes(*k*[,*Opt1*[,*Opt2*]])	Requests ***k*** passes
-specific(*op1*[,*op2*])	Options ***op1*** before and ***op2*** after every specific walk
-specific_climit(*n*)	Per function Limit on the no. of recorded calls
-specific_wlimit(*n*)	Walk limit on the number of recursively generated calls
-specific_retry(*n*)	**n**== 0 implies inhibiting rewalking with same parameters
-static_depth(*n*)	Adjusts the depth of static variable analysis

.... value tracking flags ....

fai pointed-to Arg is Initialized(ON)
fii Inhibit Inference (OFF)

fiw Initialization is a Write (ON)
fiz Initialized by Zero is a Write (ON)
fnn (C++) **new** can return NULL (OFF)
fnr Null ptr may be Returned (OFF)
fpn Pointer param may be NULL (OFF)
fsp SPecific function calls (ON)
fsv track Static Variables (ON)

---- Miscellaneous Detection Options ----

-deprecate(*category*,*name*,*commentary*) Deprecates use of a ***name***; categories: **function**, **keyword**, **macro**, **variable**
+headerwarn(*file*) Causes Msg **829** to be issued for a given ***file***
-idlen(*n*[,*opt*]) Report identifier clashes in ***n*** chars
opt: **x**=external, **p**=preprocessor, **c**=compiler
-size(*flags*,*amount*) Report large aggregates; ***flags***: **a** auto, **s** static

.... special detection flags ....

fet requires Explicit Throws (OFF)
fil Indentation check of Labels (OFF)

_____ MISC. GROUP _____

---------- Global Options ----------

-background Reduces task priority
-/**+**/**++b** No/Redirect/Produce Banner line
-p[**(*n*)**] Just preprocess, ***n*** == max output width
-setenv(*name*=*val*) Sets an environment variable
-u Unit checkout

.... global flags ....

fpa PAuse before exiting (OFF)

---------- File Options ----------

-/+cpp(*extension*) Remove/add .ext for C++ files
+ext(*ext*[,*ext*]...) Extensions attempted for extensionless files; defaults to **+ext(vac,lnt,cpp,cxx,c)**
-indirect(*file*) Process indirect (**.lnt**) file
+libclass([**all,angle,ansi,foreign**]...**)** Default library headers
-/+libdir(*directory*[,...]) Deny or specify library directory
-/+libh(*header*[,...]) Deny or specify library header by name

-/+libm(*module*[,...])	Deny or specify library module by name
-library	Sets library flag
+lnt(*ext*)	file**.*ext*** is treated like file**.lnt**
-maxfiles(*n*)	Sets an upper limit on the number of files
-maxopen(*n*)	Assumed number of openable files
-pch(*hdr*)	Designates ***hdr*** as the pre-compiled header
-subfile(*indirect-file*, options|modules)	Process just **options** or just '**modules**' from indirect-file
--u	**-u** and ignore modules at lower **.lnt** level

.... file flags ....

fcp (C++) Force C++ Processing (OFF)	**fda** Dbl-qts to Angle brackets (OFF)
fdh append '**.h**' to Header names (OFF)	**fdi** Directory of Including file(OFF)
flb Library (OFF)	**frb** Files **fopen**'ed with "**rb**" (OFF)
frn Treat CR as new-line (OFF)	**fsh** SHared file open (OFF)
ftr TRuncate filenames to 8x3 (OFF)	

--------- Output Options ---------

-od[***options***]**(*filename*)**	Output declarations of defined external objects and functions including prototypes, options: **f**=only functions, **i**=internal functions, **s**=structs, ***integer***=specify break width
-oe(*filename*)	Redirect to standard Error (**+oe** appends)
-ol(*filename*)	Output library file
-oo[**(*filename*)**]	Output to lint object file
-lobbase(*filename*)	Establish a lob base file
-os(*filename*)	Redirect to standard out (**+os** appends)
+program_info(output_prefix=*prefix*, *suboptions*)	Dumps information into
***prefix*file.txt**	information about files
***prefix*symbol.txt**	information about symbols
***prefix*type.txt**	information about types
***prefix*macro.txt**	information about macros
+stack(*sub-options*)	Issue report on function stack usage
&file=*filename*	Designates name of file to receive stack report
&overhead(*n*)	Sets the overhead of each function call
&external(*n*)	Sets assumed stack usage by each external function
name(*n*)	Sets stack usage **(*n*)** explicitly for named function
&summary	Requests just a summary of stack info report.

.... output option flags ....

fod Output Declared to object (OFF)	**fol** Output Library to object (OFF)

Appendix B

Print Utility for PC-lint

Many users of PC-lint already have editors and/or utilities that will display a file's line numbers. However, just in case you don't have one, we have included such a utility, **pr.exe**, on the PC-lint distribution diskette.

The program will both print or write (to a file) line-numbered files. It has a large number of options to handle a wide variety of printers and chores. See the examples below.

The program attempts to satisfy two schools of thought with respect to where the formfeed button should position the paper. The first school believes that the crease should be under the print head whereas the second school believes that formfeed should advance the paper to the point where it can be torn off easily. In its default configuration, **pr** is set up for the second school; it starts printing immediately and then adds either a formfeed or additional blank lines (if **-w** option) to complete a page. Both the number of blank lines and the page length are parameters. The first school (the crease mavens) can select the **-c** option to initially place a number of blank lines on the page and this number is deducted from the number of blank lines that would ordinarily be appended.

For example if the page length was 66 (**-l66** option, which is the default) and if the page separation (slightly misnamed bottom margin) is set to 6 (**-l6** option, which is the default) then a 3 line header is printed followed by 57 lines from the file to be printed and than either a formfeed or 6 blank lines to get to the next page. If the **-c** (crease option) were given, 3 blank lines would precede the 3 line header. After printing 57 lines, either a formfeed or a 3 blank line trailer would be printed.

Usage

pr [***options***] [***file*** ...]

paginates and prints the sequence of files onto the printer (or standard out) as modified by the ***options*** indicated below.

The files may contain wild card characters ***** and **?**.

-b# Sets the bottom margin to # (see above).

-c The printer is starting at a crease in the paper. See above.

-d double space

-d3 triple space, etc

-h removes headers and footers from output.

-i# request an indentation of each line and header and footer by # tabs. (**-i** means one tab).

-l# specifies the # of lines per page (66 is assumed). If the # is less than twelve, no headers are produced (equivalent to the **-h** option).

-n suppresses line numbering.

-s diverts output to the standard output file (otherwise output goes to the printer).

-t# specifies the width of a tab character. The default is 8.

-w Use white space rather than form-feeds to separate pages.

+Page Page is a Starting Page number; if omitted, 1 is assumed.

-Page Page is an Ending Page number; if omitted, the last page is assumed.

- input comes from standard input (Note that this may be included among the files).

Options **-b#**, **-c**, **-d**, **-h**, **-l#**, **-n**, **-s** and **-t#** apply to the entire run and for your convenience need not precede the files. Options ***+Page*** and ***-Page*** apply only to the file immediately following. Exception: if ***+Page*** and/or ***-Page*** are followed by no file they are applied to all files. All formfeeds within a file are taken as pagebreaks.

In addition, issuing **pr** without arguments gives a brief help message.

Examples

```
pr alpha.c
```

prints **alpha.c** paginated and line numbered on the printer (**PRN:**).

```
pr alpha.c -s -h | more
```

displays **alpha.c** on the screen with line numbers (one screenful at a time).

```
pr  alpha.c -b0 -l60
```

prints **alpha.c** on a laser printer. (It is assumed that the laser printer has a page length of 60 and that white space needn't separate pages.) The **-w** inhibits formfeed after the 60th line.

```
pr alpha.c -5 beta.c
```

prints all of **alpha.c** and the first five pages of **beta.c** on the printer

```
pr -l1 +100 -110 foo
```

prints lines numbered 100 through 110 from file **foo**. Note that the **-l1** option eliminates headers as well as setting the page length to one.

```
pr huge -l1 +10000 -n -s >temp
```

extracts from file **huge** lines 10000 onward placing them in file **temp**.