

Reference Manual

for

PC-lint/FlexeLint

A Diagnostic Facility

for

C and **C++**

Software Version 8.00 and later
Document Version 8.00

July, 2001

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

www.gimpel.com

Copyright (c) 1985 - 2001 Gimpel Software
All Rights Reserved

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without the express written permission of Gimpel Software.

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	1
2. INTRODUCTION	2
2.1 An Example	2
2.2 A Lint for C++	3
2.3 Language Definition	4
3. GETTING STARTED WITH PC-LINT	6
3.1 Setup	6
3.2 Configuration	7
3.3 Running the Test Programs.....	8
3.4 Linting your Programs	10
3.4.1 Other Extents for C++ Modules	11
3.4.2 Controlling the Messages.....	11
3.4.3 Options	12
3.4.4 Extending the Command Line.....	12
3.5 Integrating With Your Environment.....	12
3.6 .dsp processing	13
3.7 Other Operating Systems	15
3.7.1 Multiple Configurations	16
3.7.2 DOS-ins Files	16
3.7.3 DOS Extender Notes.....	18
4. THE COMMAND LINE	20
4.1 Indirect (.Int) Files	21
4.2 Exit Code.....	21
5. OPTIONS.....	23
5.1 Rules for Specifying Options.....	23
5.2 Error Inhibition Options.....	26
5.3 Size and Alignment Options	38
5.4 Verbosity Options	40
5.5 Flag Options.....	42
5.6 Message Presentation Options	59
5.6.1 Message Height Option	60
5.6.2 Message Width Option	62
5.6.3 Message Format Options	62
5.6.4 -append.....	63
5.7 Other Options.....	64
5.8 Compiler Adaptation.....	84
5.8.1 Microsoft Keywords.....	86
5.8.2 Compiler Codes.....	87
5.8.3 Customization Facilities	90
5.8.4 Identifier Characters.....	94
5.8.5 Preprocessor Statements.....	95
5.8.6 In-line assembly code	95
5.8.7 The General Solution	96
6. LIBRARIES	97
6.1 Library Header Files.....	97
6.2 Library Modules	101

6.3 Library Object Modules	103
6.4 Assembly Language.....	104
7. LINT OBJECT MODULES	106
7.1 What is a LOB?.....	106
7.2 Why are LOB's used?	106
7.3 Producing a LOB.....	108
7.4 -lobbase to reduce lob sizes.....	108
7.5 Make Files.....	110
7.6 Library Modules	111
7.7 Options for LOB's	112
7.8 Limitations of LOB's.....	112
8. STRONG TYPES	113
8.1 Quick Start	113
8.2 What are Strong Types?.....	113
8.3 -strong	114
8.4 -index	118
8.5 Type Hierarchies.....	120
8.5.1 The Need for a Type Hierarchy	120
8.5.2 The Natural Type Hierarchy	122
8.5.3 Adding to the Natural Hierarchy	123
8.5.4 Restricting Down Assignments (-father)	124
8.5.5 Printing the Hierarchy Tree.....	125
8.6 Hints on Strong Typing.....	126
8.7 Reference Information	127
8.8 Strong Types and Prototypes	129
9. VALUE TRACKING	130
9.1 Initialization Tracking.....	130
9.2 Value Tracking	133
9.2.1 The assert remedy	136
9.2.2 Interfunction Value Tracking.....	137
9.2.2.1 Specific Calls - Specific Walks.....	140
9.2.2.2 Specific Call Portrayal	141
9.2.2.3 Summary of Options	143
10. SEMANTICS	148
10.1 Function Mimicry (-function)	148
10.1.1 Special Functions	148
10.1.2 Function listing	152
10.2 Semantic Specifications.....	157
10.2.1 Possible Semantics	158
10.2.2 Semantic Expressions.....	161
10.2.3 Notes on Semantic Specifications.....	166
11. OTHER FEATURES	167
11.1 Order of Evaluation	167
11.2 Format Checking.....	168
11.3 Indentation Checking.....	168
11.4 const Checking.....	170

11.5	volatile Checking	171
11.6	Prototype Generation	171
11.7	Exact Parameter Matching	173
11.8	Weak Definials	175
11.8.1	Unused Headers.....	176
11.8.2	Removable from Header.....	177
11.8.3	static-able	178
11.9	Unix Lint Options.....	178
11.10	Static Initialization	179
12.	NON-ANSI EXTENSIONS	180
12.1	Memory Models.....	180
12.2	Additional Reserved Words	181
13.	PREPROCESSOR	182
13.1	Preprocessor Symbols.....	182
13.2	include Processing	182
13.2.1	INCLUDE Environment Variable.....	183
13.3	ANSI Preprocessor Facilities	184
13.3.1	#line and #.....	184
13.4	Non-ANSI Preprocessing	184
13.4.1	#assert	184
13.4.2	#c_include	185
13.4.3	#asm.....	185
13.4.4	#dictionary	185
13.4.5	#endasm	185
13.4.6	#import.....	186
13.4.7	#unassert	186
13.4.8	#include_next.....	186
13.5	User-Defined Keywords	187
14.	LIVING WITH LINT	188
14.1	An Example of a Policy	188
14.2	Recommended Setup	190
14.3	Using Lint Object Modules.....	191
14.4	Summarizing	192
15.	COMMON PROBLEMS	193
15.1	Option has no effect.....	193
15.2	Order of option processing	193
15.3	Too many messages.....	193
15.4	What is the preprocessor doing?	194
15.5	NULL not defined.....	194
15.6	Error 123 using min or max	195
15.7	LONG_MIN macro.....	195
15.8	Plain Vanilla Functions	195
15.9	Avoiding Lint Comments in Your Code	196
15.10	Strange Compilers.....	196
15.11	!0	197
15.12	What Options am I using?	197

15.13	How do I deal with SQL?	198
15.14	Torture Testing Your Code	198
16.	ADDITIONAL NOTES	199
16.1	Size of Scalars	199
17.	MESSAGES	201
17.1	C Syntax Errors	204
17.2	Internal Errors	216
17.3	Fatal Errors	216
17.4	C Warning Messages	218
17.5	C Informational Messages	249
17.6	C Elective Notes	269
17.7	C++ Syntax Errors	279
17.8	C++ Warning Messages	286
17.9	C++ Informational Messages	296
17.10	C++ Elective Notes	308
18.	WHAT'S NEW	316
18.1	Major New Features	316
18.2	New Error Inhibition Options	318
18.3	New Verbosity Options	318
18.4	New Flag Options	319
18.6	Additional Other Options	319
18.7	Compiler Adaptation	320
18.8	New Pre-Processor Directives	321
18.9	New Messages	321
19.	BIBLIOGRAPHY	324
	Appendix A	327
	Appendix B	336

1. BASIC FACTS

The software described in this manual is distributed in one of two ways. For the PC market (Windows, MS-DOS, OS/2) 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 K&R [1] and ANSI [2] as the assumed standard for C with historical allowances for K&R C [1]. It uses the ANSI/ISO committee document [10] as the standard for C++. A number of features of the C '99 standard [4] have been incorporated in the product but we do not, at this date, support all of C '99.

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 available memory for each 2000 lines of header files.

The PC-lint package includes three executables: a Windows executable, a DOS-extended executable and an OS/2 32-bit executable. The Windows executable runs as a 32-bit console application under Windows. The DOS extended executable runs under MS-DOS and uses 80386 DOS extender technology to access all the extended memory available. The OS/2 32-bit executable runs only under OS/2.

The FlexeLint package consisting primarily of C source code 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 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.

Several navigational aides may not be immediately obvious. Both the Table of Contents and the Index are clickable. Thus, if you find an index item you can click on each of the page numbers to take you rapidly to the indicated page.

To get back from your excursion, you need to use the relatively non-intuitive:

Control + Left-Arrow.

This is, of course, documented and it is even a menu item (Document, Go Back). However, for folks coming from a web browser, where backing up is blatantly obvious, this subtlety deserves pointing out.

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 multiplication in line 10 unexpectedly loses precision even if `long` are longer than `int`; 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'être 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 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 10 years ago, lint tools too 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 229 warnings and 162 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 8. 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 9.1 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 9.2 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 10.1 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 dos 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 as well as a thorough description of current practice within the C community 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 [\[2\]](#) 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]. These efforts led to an international standard for C++ [10].

For C modules, PC-lint/FlexeLint assumes the C89 definition of C and supports K&R where it does not conflict with C89. It also supports common extensions to the standard especially for specific compilers. See **Chapter 12. NON-ANSI EXTENSIONS, Section 5.8 Compiler Adaptation** and **Chapter 13. PRE-PROCESSOR** for more information on common extensions. We are gradually incorporating parts of the C99 standard largely on an as-need basis. For C++, PC-lint/FlexeLint is fully supporting the international standard [10].

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 and Windows 2000. For other operating systems see [Section 3.7 Other Operating Systems](#).

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 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:

<code>lint-nt.exe</code>	PC-lint executable.
<code>config.exe</code>	Configuration file for PC-lint.
<code>readme.txt</code>	Addendum to the manual.
<code>pc-lint.pdf</code>	On line manual in Portable Data Format.
<code>pr.exe</code>	A convenient printing utility.
<code>msg.txt</code>	ASCII rendition of our message descriptions.
<code>unwise.exe</code>	An uninstall program.
<code>install.log</code>	An installation log.
<code>Lint\</code>	A subdirectory containing ...
<code>co-....lnt</code>	Compiler options files for specific compilers.
<code>co.lnt</code>	A generic compiler options file.
<code>sl-....c</code>	Standard library modules for non-ANSI compilers.
<code>sl.c</code>	A generic standard library module for non-ANSI compilers.
<code>env-....lnt</code>	Options files for various environments including Microsoft's Visual Studio and various editors.
<code>lib-....lnt</code>	Options files for special 'challenge' libraries.
<code>au-....lnt</code>	Options files for author recommended checks.
<code>Test\</code>	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 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 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 be run before linting, which will 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 14.2 Recommended Setup.](#)

To summarize, the wizard will create the following files:

LIN.BAT	A 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	A 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 14.2 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 appear:

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 diskette 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. 8.00, Copyright ...

--- Module: filea.cpp

--- Module: fileb.cpp

```
    _  
};  
fileb.cpp 4 Warning 631: tag X defined differently at  
           line 3, file filea.cpp
```

--- Global Wrap-up

**Warning 1527: static member X::a (line 5, file filea.cpp)
not defined**

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 any). 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, Infor-

mational, 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 17. MESSAGES](#)).

In many cases (including this example) a prior point in another module 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).

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. You will note in particular Message [832](#) which is issued for the sole purpose of allowing an automatic error processor to go to the location cited within the immediately preceding message as described in the previous paragraph.

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 an output file on the distribution disc named:

```
test?.out
```

To run the first of these type at command level:

```
lint 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 `.int` 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 Extents for C++ Modules

To override the default treatment of extents 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 14. 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:

```
alphabet.lnt:
    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 14.2 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. At this writing this list includes:

<u>File</u>	<u>Environment</u>
<code>env-BC5.lnt</code>	Borland's 5.0 IDE
<code>env-bwin.lnt</code>	Borland's 4.0 IDE under Windows
<code>env-cw.lnt</code>	Premia Codewright
<code>env-pwb.lnt</code>	Microsoft's Programmer's Work Bench
<code>env-si.lnt</code>	Source Insight Editor
<code>env-sled.lnt</code>	Visual Slick Edit
<code>env-tide.lnt</code>	Turbo/Borland IDE under DOS
<code>env-vc.lnt</code>	Microsoft's Visual C/C++, Version 1
<code>env-vc2.lnt</code>	Microsoft's Visual C/C++, Version 2
<code>env-vc4.lnt</code>	Microsoft's Visual C/C++, Version 4
<code>env-vc5.lnt</code>	Microsoft's Visual C/C++, Version 5

These files are updated as needed. See our web site (under Version 8.00 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 .dsp processing

If you are using Microsoft Visual Studio, then an abstract of each project is retained in a file whose extension is `.dsp`. This file is in ASCII and can be viewed with your editor. The file contains module names, defines of pre-processor variables, and include paths. PC-lint/FlexeLint is able to process such files, extract such information as would be related to linting, transform the information to lint directives and out-

put the result to standard output. The resulting file, if it has a `.lnt` extension, can be used as input to PC-lint/FlexeLint. For example, the following creates a lint project file from a `.dsp` file.

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

The `.dsp` file is, among other things, inspected for `SOURCE=` lines. These lines contain not only names of modules (C and C++ files), but headers, resource files, text files, etc. PC-lint/FlexeLint, of course, wants just the module names. By default, we select out any file whose extension is `.c` and any file that bears a C++ extension. By default, C++ extensions are `.cpp` and `.cxx` but this list of extensions can be controlled via the `-cpp` option. Note that if a programmer had an unusual C extension it could be dubbed a C++ extension just for the purpose of file extraction. This gives the user complete control over which files are selected as modules.

To gain some insight into how the translation is being done, you may use the verbosity option to trigger a message for each line. Thus:

```
lint-nt -v1 project.dsp >project.lnt
```

will provide a line-by-line account of the translation.

Within a `.dsp` file, there are conditional statements such as

```
!IF "$(CFG)" == "project - Win32 Release"
```

which allows you to select debug or release variants. Variable `CFG` is generally specified to the most recent configuration within the `.dsp` file. You can change the flow via a "command line" option:

```
+d"CFG=project - Win32 Release"
```

This is like a pre-processor define except that the `+d` (as opposed to `-d`) prohibits subsequent redefinition. The use of quotes is necessary because the define contains blanks. Since there are a variety of ways of parsing command line parameters (batch command, make file, command line) and since the ways of dealing with quotes and blanks are all slightly different, it is best to place this option in a separate `.lnt` file such as:

```
release.lnt:
```

```
+d"CFG=project - Win32 Release"
```

and then use a command line such as:

```
lint-nt release.lnt project.dsp > project.lnt
```

As indicated above, the output file (`project.lnt` in these examples) 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.

3.7 Other Operating Systems

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 Windows 9X, Windows 2000, Windows ME, and Windows NT you will be using FlexeLint. For 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.BATS` and need to know how to modify each one of them and not have the current one blindly modified.

The batch files created are:

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

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

<code>LIN.BAT</code>	(<code>lin.cmd</code> on OS/2) A batch file, which <code>INSTALL</code> will copy into your <code>PATH</code> and which contains a reference to <code>STD.LNT</code> .
<code>STD.LNT</code>	An indirect lint file that will "point to" a compiler options file and an options file.
<code>STD_...LNT</code>	Possibly multiple versions of <code>STD.LNT</code> (See Section 3.7.1 Multiple Configurations.)
<code>OPTIONS.LNT</code>	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 14.2 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 14.2 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 Windows 9X, Windows ME, Windows NT, and Windows 2000 operating systems. 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 17. MESSAGES).

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

```
DOS-ins\lint
DOS-ins\test
DOS-ins\pharlap
```

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

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

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 NT and Windows 95. 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 DPML. 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 (**.int** files). Indirect files may contain more options and files. [See Section 4.1 Indirect \(.int\) 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 **.int** 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 13.2 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 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. For Macintosh systems the characters `'{'` and `'}'` are used instead of `'%'` and `'%'`.

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

```
lin.bat:

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:

```
std.lnt:

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(#)** and see [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. Note that the 'lint' within the comment must be lower-case as shown and must lie immediately adjacent to the '/' or '//'. The */*lint* comment may span multiple lines. Note that within indirect files (.lint files), options 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 11.9 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, ... )
```

Whereas these options can always be placed in `.lint` indirect files and in `/*lint` comments, they sometimes conflict with command interpreter meta characters and cannot be placed on the command line. Alternative characters are then called for.

In particular, for Unix, use square brackets (`[` and `]`) as an alternative to parentheses as in the command:
`lint -esym[714,alpha]..`

For MS-DOS, for passing parameters to a batch file, use `!` as an alternative to comma (`,`). Thus, although the command line

```
c:\lint\lint -esym(714,alpha) ...
```

may work you will need to modify the option when invoking `lin` (the batch file) as follows:

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

The full set is:

<i>O.S.</i>	<i>Meta Character</i>	<i>Alternative</i>
MS-DOS	,	!

Unix	() ?	[] .
Macintosh MPW	+ "	& '

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 output 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 14. 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 the 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 `-elib{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 x` 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{???}` or `-e{*}` are legitimate.

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

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 C code, the `/*lint` form of comment would be more appropriate:

```

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

```

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 'l'. If this option could be embedded in a macro, such a rapid search could not be done.

-ealetter 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**.

-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*.

-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 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`.

`-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()`.

`-elibsym(# [, #] ...)` inhibits

`+elibsym(# [, #] ...)` re-enables

the numbered messages 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 */
```

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.

`-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 sup-
 pressed 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.

`-epletter(s)` Pointer to Type Mismatch

This option refers to pointer-to-pointer mismatch (Error 64) across assignment or implied assign-
 ment 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.
- epb** 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**. -**epb** 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.

-**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**. Only messages that are parameterized by the identifier *Symbol* or *Name* can be so suppressed. Thus, if you examine Message **714** in **Section 17.5 C Informational Messages** you will notice that the italicized word '*Symbol*' appears in the text of the message. See also the entry for '*Symbol*' in the beginning of **Chapter 17. 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`.

Arguments (both error number and symbols) may contain 'Wild-card' characters as follows:

```

'*'    any number of characters
'?'    any single character

```

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 1 character and for any name containing a "::", i.e. for any member name. An another example:

```

-esym( x, name )

```

suppresses any message about symbol name.

Although an option containing wild card 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.

The error inhibition of `-esym` is independent of the `-e#` option. For a message regarding a particular symbol to be reported, its number must not be inhibited by `-e#` and it must not be inhibited by `-esym(#,Symbol)`. For example, the combination

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

does *not* enable message **714** for **alpha**. The first option suppresses **714** completely independently of any **esym** option. The second option, unless there was a prior **-esym(714,alpha)**, has no effect.

-etd(*TypeDiff* [, ...]) inhibits

+etd(*TypeDiff* [, ...]) re-enables

messages arising through certain specified type differences. **Chapter 17. 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.

Also of interest ...

-wLevel 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 *PARAM*. 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.

For every size option of the form:

-sType #

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

-aType #

For example, the option

-ai 1

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 sizes) may be needed. An attempt to use the **-aType** option (or the **-sType** option) after this point is greeted with Error [75](#).

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[cdhoistw][mf<int>]`. There may be zero or more characters chosen from the set "cdhoistw#". 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.

-vc... The 'c' flag may be added to the verbosity option. 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 9.2.2 Interfunction Value Tracking.](#)

-vd... We have added a Dialect ('d') flag to the verbosity option (-v). The Dialect flag ('d') will cause a language dialect identifier (either 'C' or 'C++') to be appended to each module verbosity message. Thus if the verbosity option is **-vfd**, then the module verbosity could resemble:

```
--- module:  alpha.cpp (C++)
```

The flag will not affect the file verbosity message.

This will allow the user to, among other things, monitor the effectiveness of the **+fcp** and the **+cpp()** options.

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

-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.](#)

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

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

-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... If the letter 's' is added to the verbose option, 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 9.2.2 Interfunction Value Tracking.](#)

- vh...** The letter 'h' requests that at termination of processing the strong type Hierarchy be dumped in an elegant tree diagram. See the example in [Section 8.5.1 The Need for a Type Hierarchy](#).
- vh-...** The 'h' verbosity flag continues to mean that the strong type hierarchy is dumped upon termination. If the 'h' is followed immediately by a '-' then the hierarchy will be compressed, producing the same tree in half the lines. See [Section 8.5.5 Printing the Hierarchy Tree](#) for an example.
- 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.

<code>+f...</code>	turns a flag ON by setting the flag to 1.
<code>-f...</code>	turns a flag OFF by setting the flag to 0.
<code>++f...</code>	increments the flag by 1.
<code>--f...</code>	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.

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** **--fas**

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.

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 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.

fcpp `c++` flag (default OFF).

Options: `+fcpp` `-fcpp` `++fcpp` `--fcpp`

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 standard. That is, whether `char` is `unsigned` or `signed` is up to the implementation.

See also the `fdc` flag.

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 ANSI C++ draft [10] and is set automatically if you turn on the ANSI 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. This option has no effect when you are processing C-only code.

fdh **dot-h** flag (default OFF).

Options: `+fdh` `-fdh` `++fdh` `--fdh`

If this flag is ON and if you `#include` a file without specifying an extension, then a `".h"` extension is assumed. For example:

```
#include "stdio"
```

would cause `"stdio.h"` to be included.

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.

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
```

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

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 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. 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) 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.

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 8.5.5 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 8.5.2 The Natural Type Hierarchy](#) and [Section 8.5.3 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 8. 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.

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 11.3 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 (rules described in [Section 16.1 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**.

fkp **K&R Preprocessor** flag (default OFF).

Options: **+fkp** **-fkp** **++fkp** **--fkp**

A number of preprocessor facilities are allowed by the ANSI C standard, 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 superceded by the notion of "Library Header Files" ([See Section 6.1 Library Header Files.](#)). It still has its uses though. For example, the out-

put 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 type `wchar_t`). The size of the wide character can be specified with the option `-sw#`. This flag is set automatically when the WATCOM compiler is specified (using the option `-cwc`).

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.

fl1 **long-long** flag (default OFF).

Options: `+fl1` `-fl1` `++fl1` `--fl1`

If the long-long flag is ON (option `+fl1`) then `long long int` (or just `long long`) is a permitted type, which results in an integral quantity nominally different and usually longer than a `long int`. The size of a `long long int` can be specified with the option `-s11#`. 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 5.6.1 Message Height Option.) 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 13.3.1 #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).

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 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.

fnt Nested Tag flag (default ON).

Options: `+fnt` `-fnt` `++fnt` `--fnt`

A NestedTag 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-ANSI non-K&R 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-ANSI and non-K&R 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.

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 9.2 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 is supported by many compilers but is now expressly forbidden by ANSI C. There are other ways to accomplish this task, such as the ANSI C # stringize operator, provided your compiler supports them. If it doesn't, set this flag ON; see also Warning [607](#) in [Section 17.4 C Warning Messages](#).

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.

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 vendors) as

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

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

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

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.

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 9.2.2 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.

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 is 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.

fttr **Truncate** flag (default OFF).

Options: **+fttr** **-fttr** **++fttr** **--fttr**

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 (**+fttr**), then there will be an attempt to open the file `'algorithm.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.

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.

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, name1, name2, ... )
```

fwc **wchar_t** is built-in flag (default OFF).

Options: **+fwc -fwc ++fwc --fwc**

If this flag is set, **wchar_t** is built-in. It has the effect of making **wchar_t** a reserved word. Historically **wchar_t** has usually been a **typedef**. However the C++ standard now requires **wchar_t** to be a basic type. Caution: if you set this flag, the older **typedef** won't work. Check your compiler header files before turning this flag ON. The underlying type for **wchar_t** can be modified by setting its size (**-sw#**) and/or signedness (**+fwu**).

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 11.7 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 11.7 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 11.7 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 11.7 Exact Parameter Matching.](#)

fz1 **sizeof-is-Long** flag (default OFF).

Options: **+fz1** **-fz1** **++fz1** **--fz1**

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

```
+fz1  ++fz1
```

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 **-fz1** 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][F][f][a][b][r][mn][m][m/M/][I]N

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

The optional **F** means that File information is *always* provided with an error message. 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 only used 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. 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).
- %m** = the message text
- %n** = the message number
- %t** = the message type (Error, Warning, etc.)
- %l** = the line number
- %c** = the column number
- %C** = the column number + 1
- %i** = the invoking function
- %(...%)** = 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 9.2.2.3 Summary of Options.](#)

5.6.4 -append

-append(errno,string)

The option

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

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.

This option does not automatically enable the indicated message. This would be done separately with, in this example, the option `+e936`.

5.7 Other Options

-A requests strictly ANSI C/C++ processing. **-A** enables Elective Note [950](#). Non-ANSI keywords (i.e., reserved words) and other non-ANSI features (such as the `//` form of comment within C modules) are reported but duly processed according to their non-ANSI meaning. A common situation is when your compiler header files are non-ANSI but you want your ANSI program to be checked. For this situation, use, in addition to **-A**, the option **-elib(950)**. Note, to *really* check your code for ANSI compliance, lint with a set of ANSI header files.

-b Suppresses the Banner line

+b Redirects the Banner line (to standard out)

(Unlike our 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.

-background The **-background** option under Windows-NT will have the effect of running PC-lint in the background. That is, it will run at a reduced priority.

-ccode 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`.

-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:

```
-dname[=value]
```

where the square brackets imply that the value portion is optional. If `=value` is omitted, 1 is assumed. If only `value` is omitted as in `-dx=` then the `value` assigned is null. For function-like macros, see option `-dname()=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.

+dname=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.

-Dname[=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.

```
+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, 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 8.5.4 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 10.1 Function Mimicry \(-function\)](#). See also **-sem** in [Section 10.2 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)**.

-idirectory Files not found in the current directory are searched for in the directory specified. There is no intrinsic limit to the number of such directories. The search order is given by the order of appearance of the **-idirectory** options. For example:

```
-i/lib/
```

can be used (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 13.2.1 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, **-idirectory** is identical to **-idirectory**.

As a special case the option **-i-** is taken as a directive to remove all of the directories established with previous **-i** options (it has no effect on those directories specified with **INCLUDE**).

-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 13.2 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 *sitype* (or *sitype* 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 8.4 -index](#).

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

specifies the class of header files that are by default treated as library headers. *identifier* can be one of:

angle	(specified with angle brackets),
foreign	(comes from a foreign directory using <code>-i</code> or the <code>INCLUDE</code> environment variable),
ansi	(one of those specified by ANSI), 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 files. *file* may contain wild card characters. For more information, see [Section 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 11.9 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.)

`+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 `.lint` 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 7. 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.

-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 only allow 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.

-mD specifies the **D** memory model (large Data, small program model). Pointers to data areas are assumed to be **far**. [See Section 12.1 Memory Models.](#)

-mL specifies the Large memory model (pointers are assumed to be **far**). [See Section 12.1 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 12.1 Memory Models.](#)

-mS specifies the Small model (pointers are assumed to be **near**). This is the default and is provided for completeness. [See Section 12.1 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 11.6 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 s1.c -ol(s1.c)
```

can be used to capture the library declarations specified within the file **s1.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 7. 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 7.3 Producing a LOB](#). Related options are `+fo1`, `+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 `.lint` file or anywhere that a lint option can be placed (b) not all systems support redirection and (c) redirection can have strange side effects ([See Section 7.5 Make Files](#)). If `+os` is used rather than `-os`, output is appended to the file. Make sure the redirection *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.

`-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 8.5.3 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 9.2.2.3 Summary of Options.](#)

`+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 13.4 Non-ANSI 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 13.4 Non-ANSI Preprocessing](#) for descriptions of non-standard pre-processing directives.

`+pragma(name,action)` adds a `pragma`

`-pragma(name,action)` removes a `pragma`

The `+pragma(name,action)` option can be used to specify an identifier (given by *name*) that will be used to trigger an *action* when the identifier appears as the first identifier of a `#pragma` statement. *Action* must be either 'on', 'off', 'once' or 'message'.

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.](#)

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.

`+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.

`+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.

`-pragma(identifier)`

This option will remove the pragma whose name is *identifier*. Thus:

```
-cmsc
-pragma(message)
```

will remove the `message` pragma while using Microsoft C/C++.

`-printf(N, name1 [, name2] ...)`

This option specifies that *name1*, *name2*, etc. are functions, which take `printf`-like formats. The format is provided in the *N*th argument. For example, `lint` is preconfigured as if the following options were given:

```
-printf( 1, printf )
-printf( 2, sprintf, fprintf )
```

For such functions, the types and sizes of arguments starting with the variant portion of the argument list are expected to agree in size and type specified by the format. The variant portion of the argument list begins where the ellipsis is given in the function declaration. See also `-scanf`,

`-wprintf`, `-wscanf` below and [Section 10.1 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 type-def 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 (all saveable `-e` options; see `-save`) to their state at the start of the last module or the last `-save` option whichever came later. This directive is typically given within a `/*lint` comment to restore the state of the error settings. Thus:

```
/*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.

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 it was really appropriate. An immediate restore is still available using the option `-restore_` (the same option with one trailing underscore).

`+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( _loadds, asm, entry )
```

adds the three reserved words shown. `_loadds` is assigned a meaning consistent with that of the Microsoft C/C++ compiler (See [Section 5.8.2 Compiler Codes](#)). `asm` is assigned a meaning consistent with that of the Turbo C/C++ compiler. `entry` is assigned no meaning; it is simply skipped over when encountered in a source statement. Since no meaning is to be ascribed to `entry`, it could just as well have been assigned a null value as in

```
-dentry=
```

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 `-cmisc` 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 `-cmisc` 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 all error inhibition settings (made by options of the form `-e...` or `+e...` except for `-esym...`, `-efile...`, and `-elib...`). This state then can be restored with a `-restore` option. The `-save` option makes sense only within a `/*lint` comment. It can be used in a recursive option inhibition setting. For example,

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

allows macro `alpha` to be used in an error inhibition 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.

-scanf(*N*, *name1* [, *name2*] ...)

This option specifies that *name1*, *name2*, etc. are functions, which take `scanf`-like formats. The format is provided in the *N*th argument. For example, `lint` is preconfigured as if the following options were given:

```
-scanf( 1, scanf )  
-scanf( 2, sscanf, fscanf )
```

For such functions, the types and sizes of arguments following the *N*th argument are expected to be pointers to arguments that agree in size and type with the format specification. See also `-printf` above.

-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 type-def 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 [,sem] ...)`

The `-sem` option allows the user to endow his functions with user-defined semantics, or modify the pre-defined semantics of built-in functions. For example, the library function `memcpy(a1, a2, n)` is pre-defined to have the following semantic checking. The third argument is checked to see that it does not exceed the size (in bytes) of the first or second argument. Also, the first and second arguments are checked to make sure they are not NULL.

To represent this semantic you could have used the option:

```
-sem( memcpy, 1P >= 3n && 2P >= 3n, 1p, 2p)
```

The details of semantic specifications are contained in [Section 10.2 Semantic Specifications](#).

`-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](#)).

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 default) no message is given

-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 9.2.2.3 Summary of Options.](#)

-specific_climit(*n*) specifies a Call limit. The calls recorded for any one function is limited to *n*. [See Section 9.2.2.3 Summary of Options.](#)

-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 9.2.2.3 Summary of Options.](#)

-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 9.2.2.3 Summary of Options.](#)

-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 8. STRONG TYPES](#).

-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.

-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](#), [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.

-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`.

`-uName` 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`.

`--uName` 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 `-uName` 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.

`-wLevel` 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 **-wLevel** 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. It will not affect C/C++ source modules. *Level* may have the same range of values as **-wLevel** 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.

-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 non-ANSI C compiler (a compiler not using prototypes in its supplied header files). For ANSI compilers, just use the appropriate compiler options file as in

lint co-xxx module1 module2 ...

(the **.lnt** is understood). For non-ANSI 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).

The compiler specific files currently provided with PC-lint are:

<code>co-aztec.lnt</code>		Manx Aztec C
<code>co-arch.lnt</code>		Archimedes
<code>co-bc.lnt</code>		Borland/Turbo C, C++
<code>co-bcos2.lnt</code>		Borland C, C++ for OS/2
<code>co-bc4.lnt</code>		Borland C, C++ 4.xx
<code>co-bc5.lnt</code>		Borland 5.0
<code>co-cb.lnt</code>		Borland C++ Builder
<code>co-bs166.lnt</code>		BSO Tasking C166
<code>co-cc386.lnt</code>		CadUL CC-386
<code>co-qc88.lnt</code>		Code Works QC88, Version 1.1
<code>co-c86.lnt</code>		Computer Innovations C86 C
<code>co-c86p.lnt</code>		Computer Innovations C86 Plus
<code>co-c86.lnt</code>	<code>sl-c86.c</code>	Computer Innovations C86
<code>co-desm.lnt</code>		DeSmet C
<code>co-dl.lnt</code>		Datalight C
<code>co-diab.lnt</code>		Diab Data C
<code>co-ec.lnt</code>		Eco-C88 C
<code>co-frank.lnt</code>		Franklin 8051 C
<code>co-gnu.lnt</code>		GNU on Linux
<code>co-hc.lnt</code>		High C, C++
<code>co-hp.lnt</code>		Hewlett Packard C/C++
<code>co-hpacc.lnt</code>		Hewlett Parkard aCC
<code>co-ht.lnt</code>		Hi-Tech C
<code>co-ibmc2.lnt</code>		IBM C/2 (16 bit)
<code>co-ibm32.lnt</code>		IBM C, C++ Set/2 for OS/2 (32-bit)
<code>co-in.lnt</code>		Intel iC-86 and iC-286
<code>co-in96.lnt</code>		Intel iC-96 (8196 and 8198)
<code>co-in386.lnt</code>		Intel iC-386
<code>co-icb.lnt</code>		Intel Code Builder
<code>co-kc166.lnt</code>		Keil C 166
<code>co-kc51.lnt</code>		Keil C 51
<code>co-lc2.lnt</code>	<code>sl-lc2.c</code>	Lattice 2.x C
<code>co-lc33.lnt</code>		Lattice 3.3 C
<code>co-lc6.lnt</code>		Lattice 6.0 C

<code>co-mc386.lnt</code>	Microtek MCC386
<code>co-mc86.lnt</code>	Microtek MCC86
<code>co-msc.lnt</code>	Microsoft C and Visual C++
<code>co-msc20.lnt</code>	Microsoft C and Visual C++ 2.0
<code>co-msc40.lnt</code>	Microsoft Visual C++ 4.0
<code>co-msc50.lnt</code>	Microsoft Visual C++ 5.0
<code>co-msc50.lnt</code>	Microsoft Visual C++ 6.0
<code>co-mswin.lnt</code>	Microsoft for Windows
<code>co-powrc.lnt</code>	Power C, Version 1.1, 2.0
<code>co-sc7.lnt</code>	Symantec C, C++ 7.0
<code>co-sc.lnt</code>	Symantec before 7.0
<code>co-sun.lnt</code>	Sun C/C++
<code>co-ti320.lnt</code>	Texas Inst. TI320 class C compilers
<code>co-tsc.lnt</code>	TopSpeed C
<code>co-tc.lnt</code>	Turbo/Borland C, C++
<code>co-unix.lnt</code>	Unix C, C++
<code>co-vage.lnt</code>	IBM Visual Age
<code>co-wc16.lnt</code>	Watcom C, C++ (16 bit)
<code>co-wc32.lnt</code>	Watcom C/C++ (32 bit)
<code>co-wcos2.lnt</code>	Watcom OS/2
<code>co-wh.lnt</code>	Whitesmith C
<code>co-ztc.lnt</code>	Zortech C, C++
<code>co.lnt</code>	Generic Compilers

If your compiler is not found in the group above, 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 "Supplementary Diskette" 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:

Windows 95/98/2000/NT

MS-DOS (including Windows 3.1)
OS/2
Xenix
QNX

Note: this list does not include Unix!

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-...1nt` files.

5.8.2 Compiler Codes

For a number of compilers in the PC environment, there is a special option `-ccode` ([See Section 5.7 Other Options](#)). For example, `-cmsc` selects the Microsoft C/C++ compiler. The `-c` option serves 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-code.1nt` then the appropriate `-ccode`, if any, is already set. For some systems there is no need for a `-c` option and none is provided. All necessary definitions are provided in a `co.1nt` file.

The preprocessor symbols defined by `-ccode` are in addition to the ANSI pre-defined symbols (described in [Section 13.1 Preprocessor Symbols](#)) and the symbol `_1nt`, which is always pre-defined to be the version number without decimal point (such as 800). 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.

<i>Option</i>	<i>Compiler and Symbols</i>
---------------	-----------------------------

<code>-caztec</code>	Manx Aztec C compiler. Enables <code>MSDOS</code> , <code>__FUNC__</code> and <code>MPU8086</code> and, for large data pointers, <code>__LDATA</code> .
----------------------	--

-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 two features unique to the Gnu compiler. The first 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.

The second 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.

Another feature is the **#include_next** preprocessor command.

-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` (or `-za`) 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` (or `-za`) is set, then `NO_EXT_KEYS` is enabled. `long char` is enabled (`+f1c`). 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.

- @ support** 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. See `co-frank.lnt` on the PC-lint distribution media for an example of its use.

`_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-frank.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-arch.lnt` on the PC-lint distribution diskette). 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 gobble 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))
```

`-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 13.4 Non-ANSI Preprocess-](#)

ing.

`-dname()`=*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

```
- char_varying()=char*
```

As another example, for VAX-11 C,

```
- d_align()=
```

may be used to blank out the effects of the **_align(k)** option. If the macro requires *n* arguments, *n*-1 commas are required between the parentheses.

`-#dname`=*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 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 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 (`-cmtsc`) 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(&)`.

`-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`
- 20 When a template definition is found, this flag will propagate the definition to its instantiations and compile them. At one time we thought this to be necessary but now we regard this as over zealousness, and we employ just-in-time instantiation.

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 13.4 Non-ANSI Preprocessing](#) for special non-ANSI 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 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
```

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 the [Section 13.2.1 INCLUDE EnvironmentVariable](#). This chapter explains how information in header files is interpar-tial.

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.

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

```
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 `-elib` or `-wlib` 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 11.8 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 C/C++ header files, viz.

<code>assert.h</code>	<code>limits.h</code>	<code>stddef.h</code>
<code>ctype.h</code>	<code>locale.h</code>	<code>stdio.h</code>
<code>errno.h</code>	<code>math.h</code>	<code>stdlib.h</code>
<code>float.h</code>	<code>setjmp.h</code>	<code>string.h</code>
<code>fstream.h</code>	<code>signal.h</code>	<code>strstream.h</code>
<code>iostream.h</code>	<code>stdarg.h</code>	<code>time.h</code>

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

Lint facilities have traditionally described libraries through the use of a Library Module. A Library Module usually begins with

```
/*lint -library */
```

or the equivalent. It is combined with other modules while linting. For example, if `s1.c` is a Library Module, we can test `module.c` for conformance by issuing:

```
lint s1.c module.c
```

The Library Module serves several purposes. For functions, the expected argument list is described. Any object declared within a Library Module is not expected to have a definition outside the module (message [526](#) is suppressed). Also it is not required that it be used (message [714](#) is suppressed).

Before the introduction of prototypes, a Library Module would contain truncated definitions as, for example;

```
double sin(x) double x; { }
```

to describe function arguments. With the introduction of prototypes, our standard library modules contained the equivalent prototypes instead:

```
double sin(double);
```

Once compilers began introducing prototypes in standard header files, it seemed silly to have a duplicate set of prototypes in the Library Module and so the Library Modules were modified to include header files as in:

```
#include <stdio.h>
#include <math.h>
etc.
```

But as the size of header files grew, the time to process the Library Module became excessive. Also the processing became somewhat redundant since the header files were being `#included` in the programmer's own modules. Beginning with release 4.00 of PC-lint/FlexeLint we recognize some headers as 'library headers' ([See Section 6.1 Library Header Files.](#)). Objects declared within these headers needn't be defined or referenced. If they contain prototypes, then the library is fully described. As a result, for ANSI C compilers, our standard library description file has been eliminated.

Their Current Role

For non-ANSI compilers, the Library Modules serve the same role as ever.

For ANSI compilers, they may be used to describe libraries for which prototypes do not exist. Also, some programmers may prefer to *not* use prototypes because they are porting to compilers that do not support them. Since prototypes in scope serve to coerce arguments, it is somewhat safer to place the prototypes in a Library Module where they play no active role.

Creating a Library Module

Assume you are provided with a graphics library `g.lib`, in object form, and a header file `g.h` describing the library. If `g.h` contains prototypes, you don't need a special Library Module. Just make sure that `g.h` is recognized by PC-lint/FlexeLint as a Library Header. [See Section 6.1 Library Header Files.](#)

If `g.h` does not contain prototypes, you can usually prepare them from a textual description of the library provided by the vendor. If you have source for the library, you can generate prototypes using the `-od` option ([See Section 11.6 Prototype Generation.](#)). For example, if files `g1.c` through `g25.c` are the 25 source modules of the library all contained in a single directory, go into that directory and run PC-lint/FlexeLint as follows:

```
lint -u g*.c -od(gproto.h)
```

This will output declarations (including prototypes) for all functions and data objects found, to the file `gproto.h`. These declarations will not ordinarily include `struct` definitions, (to obtain them you would use `-ods`). Create a library description module `glib.c` consisting of:

```
glib.c:

/*lint -library */
```

```
#include "g.h"
#include "gproto.h"
```

Presumably `g.h` will contain the necessary structure declarations. You may then check `program.c` by issuing the command

```
lint co glib program
```

where `co` is your compiler options file. To reduce processing time you may go on and compress `glib.c` into a Lint Object Module; issue the command

```
lint -u co glib -oo(glib.lob)
```

then use

```
lint co glib.lob program
```

to check your program.

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 7. 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:

```
g.lnt:

-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 7. LINT OBJECT MODULES](#).

6.4 Assembly Language

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 14.2 Recommended Setup](#), then `std.lnt` will now have the contents:

```
std.lnt:

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.1` for example. We use the '1' extension to denote that it is seen only by lint, not by the compiler. It can contain, for starters:

```
asm.1:

/*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. LINT OBJECT MODULES

Please Note. Lint Object Modules are used solely to improve the speed of processing previously linted modules and the speed improvements are only noticeable with very large files. They do not offer any additional error checking. In fact some error checking is lost as the Lint Object Modules do not retain all the information present in the original source code.

7.1 What is a LOB?

A *Lint Object Module* is a summary (in binary form) of all 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:

```
alpha.c:

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.

7.2 Why are LOB's used?

Lint Object Modules are used to speed up the processing of 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 linked 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.

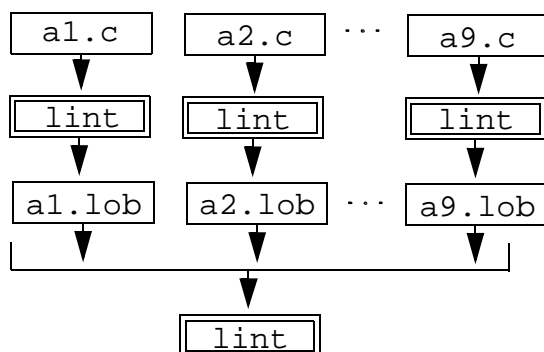


Figure LOB-1

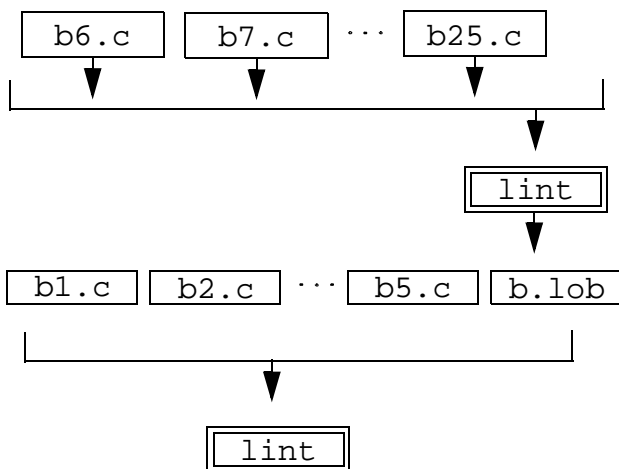


Figure LOB-2

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.

7.3 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`.

7.4 -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 "diffed" (transparently) against the base. The resulting **c2.lob** is considerably smaller (perhaps in the order of 40,000 bytes). 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 7.5 Make Files](#) to see how these dependencies can be automated.

7.5 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

```
make.lnt:

    -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 14. 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

```

make.lnt:

-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.

7.6 Library Modules

Library modules (see also [Section 6.2 Library Modules](#)) are used to describe libraries and are usually for non-ANSI compilers. They contain the option `/*lint -library*/` or its equivalent. These can also be in object form. For compilers that support ANSI 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 `s1.c`. For large library modules it makes sense to produce an object version of this module. The command

```
lint co.lnt s1.c -oo
```

(where `co.lnt` is a compiler options file) will produce the file `s1.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 `s1.c` contains:

```

s1.c:

/*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.

7.7 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. `+fo1` 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.](#)

7.8 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.

8. STRONG TYPES

*Strong type checking is gold
Normal type checking is silver
And casting is brass*

8.1 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**'ed 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)
```

8.2 What are Strong Types?

Have you ever gone through the trouble of **typedef**'ing types 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

standard 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 8.8 Strong Types and Prototypes.](#)

8.3 -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`'ed 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. (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 (`>` `>=` `<` `<=`).
 - o** ignore the Other binary operators, which are the five arithmetic 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.

By 'ignoring' we mean that no message is produced. If, for example, **Meters** is a strong type then it might be appropriate to check only Equality and Relational operators and leave others alone. In this case **JO** would be appropriate.

- 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 8.5 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 '1' flag below.

- 1** 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( AJX1, 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 **1** 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)**.

8.4 -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( AzJX, 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];
```

8.5 Type Hierarchies

8.5.1 The Need for a Type Hierarchy

Consider a *Flags* type, which supports the setting and testing of individual bits within a word. An application might need several different such types. For example, one might write:

```

typedef unsigned Flags1;
typedef unsigned Flags2;
typedef unsigned Flags3;

#define A_FLAG (Flags1) 1

```

```
#define B_FLAG (Flags2) 1
#define C_FLAG (Flags3) 1
```

Then, with strong typing, an **A_FLAG** can be used with only a **Flags1** type, a **B_FLAG** can be used with only a **Flags2** type, and a **C_FLAG** can be used with only a **Flags3** type. This, of course, is just an example. Normally there would be many more constants of each *Flags* type.

What frequently happens, however, is that some generic routines exist to deal with *Flags* in general. For example, you may have a stack facility that will contain routines to push and pop *Flags*. You might have a routine to print *Flags* (given some table that is provided as an argument to give string descriptions of individual bits).

Although you could cast the *Flags* types to and from another more generic type, the practice is not to be recommended, except as a last resort. Not only is a cast unsightly, it is hazardous since it suspends type-checking completely.

8.5.2 The Natural Type Hierarchy

The solution is to use a type hierarchy. Define a generic type called **Flags** and define all the other *Flags* in terms of it:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef Flags Flags3;
```

In this case **Flags1** can be combined freely with **Flags**, but not with **Flags2** or with **Flags3**.

This depends, however, 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 8.5.4 Restricting Down Assignments \(-father\)](#)).

A generic *Flags* type can be useful for all sorts of things, such as a generic zero value, as the following example shows:

```
//lint -strong(AJX)
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
#define FZERO (Flags) 0
#define F_ONE (Flags) 1

void m()
{
    Flags1 f1 = FZERO;           // OK
    Flags2 f2;

    f2 = f1;                     // Warn
    if(f1 & f2)                  // Warn because of J flag
        f2 = f2 | F_ONE;        // OK
    f2 = F_ONE | f2;             // OK Flag2 = Flag2
    f2 = F_ONE | f1;             // Warn Flag2 = Flag1
}
```


Note that the type of a binary operator is the type of the most restrictive type of the type hierarchy (i.e., the child rather than the parent). Thus, in the last example above, when a `Flags` OR's with a `Flags1` the result is a `Flags1`, which clashes with the `Flags2`.

Type hierarchies can be an arbitrary number of levels deep.

There is evidence that type hierarchies are being built by programmers even in the absence of strong type-checking. For example, the header for Microsoft's Windows SDK, `windows.h`, contains:

```
...
typedef unsigned int    WORD;
typedef WORD            ATOM;
typedef WORD            HANDLE;
typedef HANDLE          HWND;
typedef HANDLE          GLOBALHANDLE;
typedef HANDLE          LOCALHANDLE;
typedef HANDLE          HSTR;
typedef HANDLE          HICON;
typedef HANDLE          HDC;
typedef HANDLE          HMENU;
typedef HANDLE          HPEN;
typedef HANDLE          HFONT;
typedef HANDLE          HBRUSH;
typedef HANDLE          HBITMAP;
typedef HANDLE          HCURSOR;
typedef HANDLE          HRGN;
typedef HANDLE          HPALETTE;
...
```

8.5.3 Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via `typedef`'s 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`'s.

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) `typedef`'s will not add hierarchical links. The only links that will be formed will be via the `-parent` option.

8.5.4 Restricting Down Assignments (-father)

The option

```
-father( Parent , Child [,Child] ... )
```

is similar to the `-parent` option and has all the effects of the `-parent` option and has the additional property of making each of the links from *Child* to *Parent* one-way. That is, assignment from *Parent* to *Child* triggers a warning. You may think of `-father` as a strict version of `-parent`.

The rationale for this option is shown in the following example.

```
typedef int FIndex;
typedef FIndex Index;
```

Here `Index` is a special Index into an array. `FIndex` is a Flag or an `Index`. If negative it 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( AcJX, FIndex, Index )
```

Then

```
FIndex n = -1;
Index i = 3;

i = n;          /* Warning */
m = i;          /* OK */
```

The safe way to convert a **FIndex** to **Index** is via a function call as in

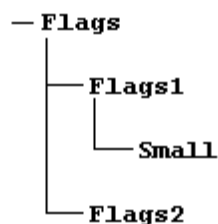
```
Index F_to_I( FIndex fi )
{ return (Index)(fi >= 0 ? fi : 0); }
```

Then, although we need to use a cast in this function we need not use a cast in the rest of the program.

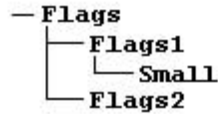
The net result of all this is that although flags and indices occupy the same storage location, we will never use a flag where an index is needed.

8.5.5 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 8.5.3 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.

8.6 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** of the previous section is an example.
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 addresses of 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 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
```

8.7 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 an indirectly referenced 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 a shift operator whose left side is a strong type.
- 6) it is a comma operator whose right side is a strong type.
- 7) it is an assignment operator whose left side is a strong type.
- 8) 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.

8.8 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.

9. VALUE TRACKING

9.1 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 10.1 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 initializa-

tion would be made:

```
class X
{
    // Warnings 1401 and 1402 are issued
    X() { g(); if( n ) n = 3; }
    void g() const;
    int n;
    int m;
};
```

9.2 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) across statements in a fashion similar to what is retained about the state of initialization. (See [Section 9.1 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 9.1 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 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.

9.2.1 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 10.1 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 10.1 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 takes precedence over your compiler's header files).

9.2.2 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.

9.2.2.1 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 flippantly 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 Walk* (*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.

9.2.2.2 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.

9.2.2.3 Summary of Options

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 (*.lint 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 also 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](#), [428](#), [613](#), [661](#), [662](#), [668](#), [669](#), [670](#), [671](#), [676](#), [774](#), [794](#), [795](#), [796](#), [797](#), [802](#), [803](#), [804](#), [807](#), [817](#), and [944](#) and all Internal and Fatal errors. Effectively, for each number not in the select group an option of `-ek`, where *k* 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...**

-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 $\mathfrak{f}(2)$, which will result in a Specific walk of $\mathfrak{f}(2)$, which will generate the Specific Call of $\mathfrak{f}(3)$, 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. SEMANTICS

10.1 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 10.2 Semantic Specifications](#) to see how to create custom function semantics.

10.1.1 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 10.1.2 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 10.1.2 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 10.1.2 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 10.1.2 Function listing](#). An option of the form

```
-function ( name(index), ... )
```

copies a single semantic into a destination or destinations. An option of the form

```
-function ( name, ... )
```

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 );           // Warning 668
}
```

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.

10.1.2 Function listing

<u>Function</u>	<u>f(0)</u>	<u>f(r)</u>	<u>f(1)</u>	<u>f(2)</u>	<u>f(3)</u>	<u>f(4)</u>	<u>f(5)</u>
__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	-	-	-	-
fflush	-	-	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	-	-

<u>Function</u>	<u>f(0)</u>	<u>f(r)</u>	<u>f(1)</u>	<u>f(2)</u>	<u>f(3)</u>	<u>f(4)</u>	<u>f(5)</u>
frexp	-	-	-	S1	-	-	-
fscanf	-	-	S1	S1	-	-	-
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	-	-	S7	-	-	-	-
operator delete[]	-	-	-	S7	-	-	-

<u>Function</u>	<u>f(0)</u>	<u>f(r)</u>	<u>f(1)</u>	<u>f(2)</u>	<u>f(3)</u>	<u>f(4)</u>	<u>f(5)</u>
perror	-	-	S1	-	-	-	-
printf	-	-	S27	-	-	-	-
putc	-	-	-	S1	-	-	-
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	-	-	-	-

<u>Function</u>	<u>f(0)</u>	<u>f(r)</u>	<u>f(1)</u>	<u>f(2)</u>	<u>f(3)</u>	<u>f(4)</u>	<u>f(5)</u>
strspn	-	-	S27	S27	-	-	-
strstr	-	S2	S28	S27	-	-	-
strtod	-	-	S27	-	-	-	-
strtok	-	S2	S26	S27	-	-	-
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`) the argument can be assumed to be true (i.e. non zero). See Section 9.2.1 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.

10.2 Semantic Specifications

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 10.1 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.

10.2.1 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 10.1.2 Function listing](#). As in [Section 10.1.2 Function listing](#) it is considered a Return semantic. See [Section 10.1 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 10.1.2 Function listing](#) as [s3](#); it is the semantic used for the `exit()` function. This also is considered a Return semantic.

ip (e.g. `3p`) the *i*th argument should be checked for null. If the *i*th argument could possibly be null this will be reported. For example:

```
/*lint -sem( g, 1p ) warn if g() is passed a NULL. */
/*lint -sem( f, r_null ) f() may return NULL. */
char *f();
void g(char *);
g( f() );      /* warning, g is passed a possible null */
```

This semantic is identical to the [s1](#) semantic described in [Section 10.1.2 Function listing](#).

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`.

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.

Note that for the purposes of these examples, we have placed the `-sem` options within lint comments. They may also be placed in a project-wide options file (`.lint` file).

`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`, `strchr`, `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.

exp a semantic expression involving the expression elements described below:

in denotes the *i*th argument, which must be integral (E.g. `3n` refers to the 3rd argument). An argument is integral if it is typed `int` or some variation of integral such as `char`, `unsigned long`, an enumeration, etc.

i may be @ (commercial at) in which case the return value is implied. For example, the expression:

```
@n == 4 || @n > 1n
```

states that the return value will either be equal to 4 or will be greater than the first

argument.

*i***p** denotes the *i*th argument, which must be some form of pointer (or array). The value of this variable is the number of items pointed to by the pointer (or in the array). For example, the expression:

2p == 10

specifies a constraint that the 2nd argument, which happens to be a pointer, should have exactly 10 items. The number of items "pointed to" by a string constant is 1 plus the number of characters between quotes.

Just as with *in*, *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 10.2.2 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: ?:

10.2.2 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 < lp, lp )
```

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 == lp || @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 be null or may not be. 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`'ed) 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 10.1.1 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 9.2 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 <= BUFLLEN )

const int BUFLLEN = 4;

class X
{
public:
    char buf[BUFLLEN];
    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 `BUFLLEN`. The byte requirements of "abcd" are 5 (including the nul character) and `BUFLLEN` 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
}

```

10.2.3 Notes on Semantic Specifications

1. Every function has, potentially, a Return semantic, a Function-wide semantic and Argument semantics for each of the arguments. 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 . 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.

11. OTHER FEATURES

11.1 Order of Evaluation

Expressions whose value depends on the order-of-evaluation are flagged with Warning [564](#). 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 by each side is compared with the set of variables accessed by the other side and a warning is issued for each common member. A variable is considered modified if it is subject to autoincrement or autodecrement or is assigned to.

Volatile variables and functions are subject to added scrutiny. [See Section 11.5 volatile Checking.](#)

11.2 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 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.](#)

11.3 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.

11.4 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.

11.5 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 11.4 `const` Checking.).

Alternatively, to turn off `volatile` checking you may use the option:

```
-dvolatile=
```

11.6 Prototype Generation

The following option is useful for migrating from K&R C to ANSI 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 `#included` 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)**.

-odwidth

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 8.8 Strong Types and Prototypes.](#)

11.7 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. (In a beta release we gave a message in both instances but this rendered the flag almost useless).

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 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 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.

11.8 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 -wl +e749 +e75? +e76? ...
```

11.8.1 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:

```
hdr.h:

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`.

11.8.2 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.

11.8.3 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.

11.9 Unix Lint Options

The following options are available for compatibility with other *lint*'s, 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. It 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 17. MESSAGES](#)).

11.10 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.

12. NON-ANSI EXTENSIONS

This chapter describes generally accepted, non-ANSI 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-ANSI preprocessor extensions are covered in [Section 13.4 Non-ANSI Preprocessing](#).

12.1 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](#).

<i>option</i>	<i>model name</i>	<i>data pointers</i>	<i>program pointers</i>	
-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.

12.2 Additional Reserved Words

Many compilers offer additional reserved words over those specified by ANSI. See [Section 5.8 Compiler Adaptation](#) for ways of coping with them.

13. PREPROCESSOR

13.1 Preprocessor Symbols

`_lint` The special preprocessor symbol `_lint` is pre-defined in case it is necessary 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.

`__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 compatible.

<code>__TIME__</code>	The current time
<code>__DATE__</code>	The current date
<code>__FILE__</code>	The current file
<code>__LINE__</code>	The current line number
<code>__STDC__</code>	Defined to be 1 if pure ANSI-compatibility is needed (- a or - za flag); else is left undefined

Compiler-dependent preprocessor symbols may also be established as described in [Section 5.8 Compiler Adaptation.](#)

13.2 include Processing

When a `#include "filename"` directive is encountered

1. there is first an attempt to **fopen** the named file. But what is the named file? If the **fdi** flag is OFF the name between quotes is used. If the **fdi** flag is ON, the name of the including file is examined to determine the directory. This directory is prefixed to *filename*. The directory of the including file is found by scanning backward for one of possibly several system-related special characters. If the **fopen** fails, we go to step 2.
2. there is an attempt to prepend (in turn) each of the directories associated with options of the form:

-idirectory

in the order in which the options were presented. If this fails we go to step 3.
3. On systems supporting environment variables, each directory in the sequence of directives specified by the **INCLUDE** environment variable is prepended to the file.
4. There is an attempt to **fopen** the file by the name provided.

If the include directive is of the form

```
#include <filename>
```

then the processing is the same except that step 1 is bypassed.

13.2.1 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, otherwise, may be used freely and are ignored.
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**.

13.3 ANSI Preprocessor Facilities

ANSI preprocessing is assumed throughout. If the K&R preprocessor flag is set (**+fkp**) the use of ANSI (over K&R) is flagged.

13.3.1 #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 **-f1n** ([See Section 5.5 Flag Options](#)). Some systems support **#** as an abbreviation for **#line**. To enable **#** use **++f1n** (increment the **f1n** flag to 2). For example,

```
//lint ++f1n 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
```

13.4 Non-ANSI 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](#).

13.4.1 #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](#).

13.4.2 #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)
```

13.4.3 #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](#).

13.4.4 #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 action required by the semantics of **#dictionary**. The **#dictionary** directive need not be specially enabled for VAX VMS, but for every other systems a **+ppw(dictionary)** would be required before use.

13.4.5 #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,0
ADD AH,AL
#endasm       /* end assembly code */
```

13.4.6 #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.

13.4.7 #unassert

The **#unassert** preprocessor directive has the same syntax as **#assert** and undoes the effects of **#assert**. This is compatible with the Unix convention.

13.4.8 #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 13.2 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
```

13.5 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.

14. 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.

The policy will be automatically imposed by incorporating the error suppression options in a `.lint` 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 15.14 Torture Testing Your Code](#).

14.1 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/Flexe Lint 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 oversized `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).

When programming in pure C, I tend *not* to call in the presence of a prototype. Calling with a prototype in scope is not yet completely portable and can cause quiet unintended conversions. Hence I routinely use option `-e746` to suppress the "not called with a prototype" message. This does not effect my C++ coding style. C++ requires that I use prototypes and a missing prototype is greeted with an error message ([1055](#)).

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          // test of assignment
-e701 -e703    // shifting int left is OK
-e718          // undeclared function
-e746          // allow calls w/o prototypes
-passes (2)    // use two passes

```

14.2 Recommended Setup

For PC-lint the INSTALL program creates a batch file `lin.bat` (`lin.cmd` on OS/2) that indirectly includes the centralized `options.lnt` file described in the previous section. This situation is depicted in *Figure Setup*. 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`.

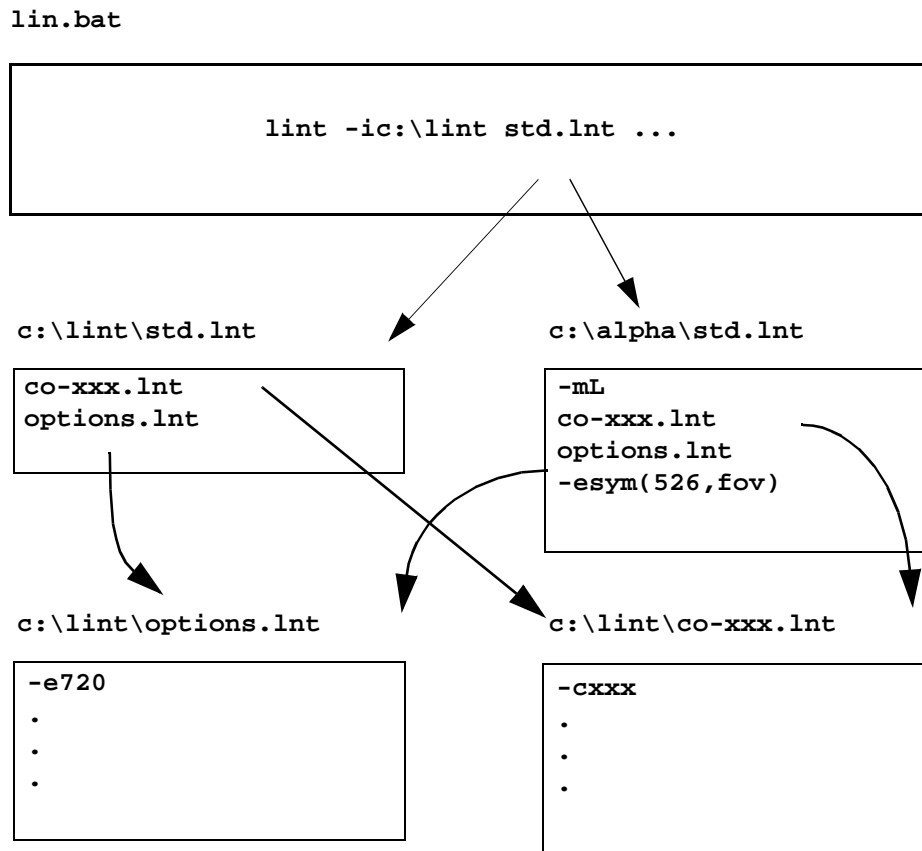


Figure Setup

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.

14.3 Using Lint Object Modules

For large projects (more than 50 source modules), I use Lint Object Modules ([See Section 7.1 What is a LOB?.](#)).

For FlexeLint my **make** file setup is similar to that described in [Section 7.5 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

```

14.4 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!

15. COMMON PROBLEMS

15.1 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;
```

15.2 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.

15.3 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 decla-

ration) 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..)

15.4 What is the preprocessor doing?

In order to understand some diagnostic 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.

15.5 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.

15.6 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)`.

15.7 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.

15.8 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`'s and `short`'s are promoted to `int`, and `float`'s are 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 C standard.

15.9 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 `.lint` 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)
```

15.10 Strange Compilers

You may want to lint programs that have been prepared for compilers that accept strange and unusual con-

structs, 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.

15.11 !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)
```

15.12 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 `?`.

15.13 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)
```

15.14 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	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	warns about the use of pointer parameters without first checking for NULL.
-strong(AJX)	All typedefs must match exactly.
-w4	Set warning to the max.

16. ADDITIONAL NOTES

16.1 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 suppresses `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 int`, `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:

<code>int</code>	<code>sizeof(int)*bits-per-byte - 1</code>
<code>unsigned int</code>	<code>sizeof(int)*bits-per-byte</code>
<code>long</code>	<code>sizeof(long)*bits-per-byte - 1</code>
<code>unsigned long</code>	<code>sizeof(long)*bits-per-byte</code>

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, 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.

17. 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 from a self-extracting executable `msg.exe`, found on the PC-lint distribution diskette, which produces the machine-readable ASCII file `msg.txt`. For FlexeLint users, the file `msg.txt` can be found on the FlexeLint Supplementary diskette.

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.

	<u>C</u>	<u>C++</u>	<u>Warning Level</u>
Syntax Errors	1 - 199	1001 - 1199	1
Internal Errors	200 - 299		0
Fatal Errors	300 - 399		0
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:

<i>argument</i>	The actual argument of a function as opposed to a dummy (or formal) parameter of a function (see parameter).
<i>arithmetic</i>	any of the integral types (see below) plus <code>float</code> , <code>double</code> , and <code>long double</code> .
<i>Boolean</i>	In general, the word Boolean refers to quantities that can be either true or false. An expression is said to be Boolean (perhaps it would be better to say 'definitely Boolean') if it is of the form: <i>operand op operand</i> where <i>op</i> is a relational (<code>></code> <code>>=</code> <code><</code> <code><=</code>), an equality operator (<code>==</code> <code>!=</code>), logical <i>And</i> (<code>&&</code>) or logical <i>Or</i> (<code> </code>). A context is said to require a Boolean if it is used in an <code>if</code> or <code>while</code> clause or if it is the 2nd expression of a <code>for</code> clause or if it is an argument

to one of the operators: `&&` or `||`. An expression needn't be definitely Boolean to be acceptable in a context that requires a Boolean. Any integer or pointer is acceptable.

<i>declaration</i>	gives properties about an object or function (as opposed to a definition).
<i>definition</i>	that which allocates space for an object or function (as opposed to a declaration) and which may also indicate properties about the object. There should be only one definition for an object but there may be many declarations.
<i>integral</i>	a type that has properties similar to integers. These include <code>char</code> , <code>short</code> , <code>int</code> , and <code>long</code> and the <code>unsigned</code> variations of any of these.
<i>scalar</i>	any of the arithmetic types plus pointers.
<i>lvalue</i>	is an expression that can be used on the Left hand side of an assignment operator (<code>=</code>). Some contexts require lvalues such as autoincrement (<code>++</code>) and autodecrement (<code>--</code>).
<i>macro</i>	an abbreviation defined by a <code>#define</code> statement. It may or may not have arguments.
<i>member</i>	elements of a <code>struct</code> and of a <code>union</code> are called members.
<i>module</i>	that which is compiled by a compiler in a single independent compilation. It typically includes all the text of a <code>.c</code> (or a <code>.cpp</code> or <code>.cxx</code> , etc.) file plus any text within any <code>#include</code> file(s).
<i>parameter</i>	A formal parameter of a function as opposed to an actual argument (see argument).

Message Parameters

Some of the messages are parameterized with one or more of the following italicized names:

<i>Char</i>	Some character
<i>Context</i>	Specifies one of several contexts in which an assignment can be made. Can be one of: <ul style="list-style-type: none">• <code>assignment</code> -- refers to an explicit assignment operator.• <code>return</code> -- refers to the implied assignment of a <code>return</code> statement. The type of the expression is converted implicitly to the type of the function.• <code>initialization</code> -- refers to the assignment implied by an initialization statement.• <code>arg. no....</code> -- 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.• <code>arg. 'this'</code> -- refers to the implied argument of a member function call.
<i>FileName</i>	A filename. Messages containing this parameter can be suppressed with the <code>-efile(...)</code> option.
<i>Integer</i>	Some integer

<i>Invocation</i>	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 <code>f(1)</code> could result in Error 1024 parameterized by ' <code>f(int)</code> '. To suppress this message you could use <code>-esym(1024,f(int))</code> . You could also use wild cards as in <code>-esym(1024,f*)</code> .
<i>Kind</i>	A list of control structures.
<i>Location</i>	A line number followed optionally by a filename (if different from the current) and/or a module name if different from the current.
<i>Name</i>	A <i>string</i> , usually an identifier, that can be suppressed with a <code>-esym</code> but not with <code>-elibsym</code> .
<i>String</i>	A sequence of characters identified further in the message description.
<i>Symbol</i>	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 <code>-esym(...)</code> option. For C++, if <i>Symbol</i> is the name of a function the full function signature (including parameters) is given. Error suppression with <code>-esym</code> does not require (nor want) the parameters.
<i>Type</i>	A type or a top type base is provided. A top type base is one of <code>pointer</code> , <code>function</code> , <code>array</code> , <code>struct</code> , <code>union</code> , or <code>enum</code> .
<i>TypeDiff</i>	<p>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 <code>-etd</code> option to inhibit errors based on type differences. <i>TypeDiff</i> can be one or more of:</p> <ul style="list-style-type: none"> • basic -- The two types differ in some fundamental way such as <code>double</code> versus <code>int</code>. • 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 15.8 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, <code>int</code> versus <code>long</code> where these are the same size or <code>double</code> versus <code>long double</code> 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 <code>long</code> is the same size as <code>int</code> then <code>unsigned long</code> will differ from <code>int</code> 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 <code>struct</code> 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 <code>-etd(origin)</code>.

- **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.

17.1 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, **#includes** 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 13.2.1 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 15.10 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 17. 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 17. 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 (0xhh)** -- 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 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 **-fssa**).
- 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**, **-epb** ([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 standard. If your compiler supports this, suppress this message.
- 82 **return <exp>; illegal with void function --** The ANSI 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.
- 90** `Option 'String' is only appropriate within a lint comment` -- The indicated option is not appropriate at the command or the `.lint` 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.

- 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.

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 for prototype` -- The number of arguments provided for a function was less than the number indicated by a prototype in scope.

- 119 **Too many arguments for prototype**-- The number of arguments provided for a function was greater than the number indicated by a prototype in scope.
- 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 15.6 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 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.
- 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 *Sym-bol*). 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 7. 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.

17.2 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.

17.3 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 7.3 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...'** -- 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` -- 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.
- 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.
- 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. 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.

17.4 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 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 Ttype` -- 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 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 9.2 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 9.2 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 9.2 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 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 9.2 Value Tracking](#).

- 417 **integral constant '*String*' 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`. *String* is the token in error.
- 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`. See [Section 10.1 Function Mimicry \(-function\)](#) and [Section 10.2.1 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 10.1 Function Mimicry \(-function\)](#) and [Section 10.2.1 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`, `memcpy`, etc.) wherein there is an apparent attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified. The function is specified by *Symbol* and the arguments are identified by argument number. See also [Section 10.1 Function Mimicry \(-function\)](#) and [Section 10.2.1 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.
- 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 than 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:

|                      |                                                                                                          |
|----------------------|----------------------------------------------------------------------------------------------------------|
| <b>Malloc</b>        | data is data obtained from a call to <code>malloc</code> , <code>calloc</code> or <code>realloc</code> . |
| <b>new and new[]</b> | data is data derived from calls to <code>new</code> .                                                    |
| <b>Static</b>        | data is either <code>static</code> data within a function or external data.                              |
| <b>auto</b>          | data is non-static data in a function.                                                                   |
| <b>Member</b>        | data is a component of a structure (and hence can't be independently freed).                             |
| <b>modified</b>      | data is the result of applying pointer arithmetic to some other pointer. E.g.                            |

```

p = malloc(100);
free(p+1); // warning

```

`p+1` is considered modified.

**constant**     data is the result of casting a constant to a pointer. E.g.

```

int *p = (int *) 0x80002;
free(p); // warning

```

**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, ln > 10 && 2n > 10)
void f(int, int);
...
 f(2, 20);
```

results in the message:

**Call to function *'f(int, int)'* violates semantic *'(ln>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 10.2.1 Possible Semantics](#).

- 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.

- 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. 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 with a terminal back-slash 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 on 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.

- 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 value** -- 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** -- 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 are larger than an `int`.
- 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 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** -- 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 in 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` [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** `Expected void type, assignment, increment or decrement.` -- The first expression of a `for` clause should either be an expression yielding the `void` type or be one of the privileged operators: assignment, increment, or decrement. See also message [522](#).
- 521** `Expected void type, assignment, increment or decrement.` -- The third expression of a `for` clause should either be an expression yielding the `void` type or be one of the privileged operators: assignment, increment, or decrement. See also message [522](#).
- 522** `Expected void type, assignment, increment or decrement.` -- If a statement consists only of an expression, it should either be an expression yielding the `void` type or be one of the privileged operators: assignment, increment, or decrement. Note that the statement `*p++;` draws this message but `p++;` does not. This message is frequently given in cases where a function is called through a pointer and the return value is not `void`. In this case we recommend a cast to `void`. If your compiler does not support the `void` type then you should use the `-fvo` option.
- 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 11.3 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** -- A portion of the program cannot be reached.
- 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 11.3 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 `\xddd` or `\xhhh` equaled or exceeded `2**b` where `b` is the number of bits in a byte (established by the `-sb` option). The default is `-sb8`.

**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 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 array 10 of `int`. 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. 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 prepro-

cessor 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 number of arguments inconsistent with format** -- The number of arguments supplied to `printf`, `sprintf`, `fprintf`, `scanf`, `fscanf` or `sscanf` was inconsistent with the number expected as a result of analyzing the format string.

**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 11.1 Order of Evaluation](#) and [Section 11.5 volatile Checking](#). Volatile variables are also checked for repeated use in an expression. See

**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 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 ANSI 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 than 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 (a possibility in the new standard), 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**'s 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 still yields **(unsigned) 1 >> 3** 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 `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 '(' --** 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.
- 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 `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 \xhex-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 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 ANSI standard. 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 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.

**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/Flex-eLint 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 8. 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 8. 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 8. 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 8. 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 8. 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 8. 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 8. STRONG TYPES](#).
- 640** `Expected strong type 'Name' in Boolean context` -- A Boolean context expected a type specified by a `-strong(B...` option. See [Chapter 8. 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 wprintf` -- 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 `wprintf`. If you are not really using `wprintf` 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 9.1 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 9.1 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) 0xFFFFu << 8      /* 648 */
```

because the `unsigned char` is promoted to `int` before shifting. The resulting quantity is actually negative. You would need to revive the `unsigned` nature of the expression with

```
(unsigned) (unsigned char) 0xFFFF << 8      /* OK */
```

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 (`+fcsu` 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 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` member.

- 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 within struct/union/enum declaration** -- 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* has no effect** -- The indicated option had no effect. Was this a mistake? Not all options are tested for effectiveness. For example, in the sequence `-w2 -e720` the 2nd option has no effect but no report is given since the nuisance value would be too high. The message is reserved for cases when the non-effectiveness would produce truly mysterious results.

- 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 9.2 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 9.2 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 ([\[22 section 20.4\]](#)) 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`. [\[22 section 20.6\]](#) 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`. See [Section 10.1 Function Mimicry \(-function\)](#) and [Section 9.2 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` option. See [Section 10.1 Function Mimicry \(-function\)](#) and [Section 9.2 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`, `memcpy`, etc) wherein there is a possible attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified.

The function is specified by *Symbol* and the arguments are identified by argument number. See also [Section 10.1 Function Mimicry \(-function\)](#) and [Section 9.2 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 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.

17.5 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).
- 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. [\[22 section 20.7\]](#) 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 11.3 Indentation Checking](#)).

726 **Extraneous comma ignored** -- A comma followed by a right-brace within an enumeration is not a valid ANSI 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 standard. 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 informa-

tional 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 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 declara-

tion 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 [Section 17.6 C 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 11.8 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 11.8 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 **typedefed** 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 11.8.3 static-able](#).
- 766 **Include of header file FileName not used in module String** -- The named header file was directly **#include**'d 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. See also Info messages [964](#), [966](#) and [Section 11.8.1 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 **++f1b ... --f1b** 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 9.1 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 20.5](#)]

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 9.2 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.

- 781** **Inconsistent use of tag 'Symbol' conflicts with Location** -- A tag specified as a **union**, **struct**, or **enum** was specified as some other type in another module (location given by *Location*). For example, if **tag** is specified as **union** in one module and is specified as **struct** in the current module you will get this message. See also Warning [407](#).
- 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 limit of 'String' 'Name' exceeded` -- Some ANSI limit has been exceeded. These limits are described in Section 2.2.4.1 of the ANSI C Standard. Programs exceeding these limits are not considered maximally portable. However, they may work for individual compilers.

Many large programs exceed the ANSI limit of 511 external identifiers. This will result in message 793 "ANSI limit of 511 '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 9.2 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 9.2 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 `+f11`) 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 effec-

tively without the `goto`. The use of `goto`'s 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`. [See Section 10.1 Function Mimicry \(-function\)](#). [See Section 9.2 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. [See Section 10.1 Function Mimicry \(-function\)](#). [See Section 9.2 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 10.1 Function Mimicry \(-function\)](#) and [Section 9.2 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*), *Con-***

text 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*' given, assumed int** -- 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 [601](#) 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.](#)).
- 815** **Arithmetic modification of unsaved pointer** -- An allocation expression (`malloc`, `cal-`

`loc, 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 ANSI.

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 can guard against their use if we 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, see what the 'conflict' is all about.

- 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.

17.6 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`.

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:

```
--emacs(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.

- 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`'s do 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 variable (either static or auto) is initialized but never modified thereafter. 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. In the case of static data it can mean that the data is ROM'able. Other situations in which a `const` can be added to a declaration are covered in messages [818](#), [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 member on *Integer* byte boundary** -- This message is given whenever padding is necessary within a struct to achieve a required member alignment. 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.

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 Required Rule *Name*, *String*** -- MISRA is the “Guidelines for the use of the C Language in Vehicle Based Software”...

The list of checks made are as follows:

- Octal constant used (Rule 19).
- Should initialize either all enum members or only the first (Rule 32).
- Side effects on right hand side of logical operator (Rule 33).
- Comma operator used outside of `'for'` expression (Rule 42).
- Null statement not in line by itself (Rule 54).
- `continue` statement should not be used (Rule 57).
- `break` used outside of a `switch` (Rule 58).
- Left brace expected for `if`, `else`, `for`, `do` and `while` (Rule 59).
- No `'else'` at end of `'if ... else if'` chain (Rule 60).
- Floating point variable used as loop counter (Rule 65).
- Function not declared at file scope (Rule 68).
- Function has variable number of arguments (Rule 69).
- Either all parameters or no parameters should have identifiers (Rule 73).
- Only preprocessor statements and comments before `'#include'` (Rule 87).
- Header file name contains non-standard character (Rule 88).
- `'#define/#undef'` used within a block (Rule 91).
- Multiple use of `'#'` and/or `'##'` operators in macro definition (Rule 98).
- Non-standard use of `'defined'` preprocessor operator (Rule 100).
- Bitfields inside union (Rule 110).

You may disable individual rules to your taste by using the `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 message 960. See [\[10\]](#) for information on the MISRA guidelines.

961 `violates MISRA Advisory Rule Name, String` -- This message is issued for some violations of the MISRA advisory guidelines.

The list of checks made are as follows:

- Constant requires numerical suffix (Rule 18)
- `'register'` class discouraged (Rule 28)
- `'sizeof'` used on expressions with side effect (Rule 40)
- Redundant explicit casting (Rule 44)
- Dependence placed on C's operator precedence (Rule 47)
- Non-case label (Rule 55)
- Boolean value in switch expression (Rule 63)
- Use of `'#undef'` is discouraged (Rule 92)

Messages can be suppressed based on rule number. See also Message [960](#).

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 11.8.1 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 11.8.1 Unused Headers](#). See also message [964](#).

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](#).

17.7 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. [\[11 section 9.4\]](#)
- 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 `.*`, `'?`, `:::` or `.'`, 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. [\[11 section 13.2\]](#)
- 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 'String(String)' 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 'Name'** -- 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 only be given 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. [\[11 section 9.4\]](#)

- 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` that 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. [\[10 section 9.4\]](#)
- 1047 **a template declaration must be made at file scope** -- A `template` declaration may not appear within a function or within a class. [\[10 section 14.1\]](#)
- 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 `arg i` for some `i` corresponds to a non-`class` parameter in the original template declaration. Such arguments need to be constants. [\[10 section 14.5\]](#)
- 1049 **Too many template arguments** -- There are more arguments in the template class-name than there were parameters in the original template declaration. [\[10 section 14.5\]](#)
- 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. [\[10 section 14.5\]](#)

- 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 consists of 2 kinds of parameters: `class identifier` and `type`. The parameter did not begin with `class` and was not a `type`. [\[10 section 14.5\]](#)
- 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. [\[10 section 14.5\]](#)
- 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 `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 [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 [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 14.5](#)]

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 14.5](#)]

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. [[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. [[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 12.5](#)].

1068 *Symbol 'Symbol' can not be overloaded --* `operator delete` or `operator delete []` can be redefined but not overloaded. There can only be one `operator delete` and one `operator delete []` but neither of these can be overloaded. [[10 section 12.5](#)].

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 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.

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 were required and it were not available. For example:

```
Class X;           // declare class X
X *p;             // OK, no definition required
X a;              // Error 1080
```

17.8 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.

- 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. According to 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.

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' has no data members` -- A variable (*Symbol*) is being instantiated that belongs to a class that contains no data members (either directly or indirectly through inheritance). [\[11 section 9\]](#)

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. [\[11 section 9.5\]](#)

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 an `auto` 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 **Data member 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 oper-

ator 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 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? [\[10 section 12.5\]](#)
- 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 check-out (-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 only intended 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 9.1 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 9.1 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 was not declared to `throw` anything. The preferred remedy is to declare the function as potentially throwing an exception. [\[23, Item 11\]](#)

1550 `function 'Symbol' may throw an exception in function 'Symbol'` -- A function was called (first *Symbol*) which was declared to throw an exception. The call was not made from within a `try` block and the function making the call was not declared to `throw` anything. Either declare the calling function as potentially throwing an exception, or place the call inside a `try` block and `catch` the `throw`.

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](#).
- 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. Uncaught exceptions should normally be declared in the function's exception specification.
- 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](#).

17.9 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 'String' 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. [\[10 section 9.4\]](#)

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.

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 **Redundant parentheses ignored** -- A vacuous initializer within a `new-initializer` for an object for which a constructor was not defined is a "syntactic convenience and has no special meaning". For example, `new double` is the same as `new double()`. [\[11 section 5.3.3\]](#)

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 these 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 mes-

sage. [\[11 section 9.3.2\]](#)

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 `Symbol`

```
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 9.1 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 9.1 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 9.1 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 not declared to 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.

17.10 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 definition, assumed '(void)'** -- 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).

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 a copy constructor's initializer list, 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. [\[12, Item 23\]](#).

- 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 initializations 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](#)]

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](#).

18. WHAT'S NEW

This chapter details the new and improved features of PC-lint/FlexeLint 8.00 over PC-lint/FlexeLint 7.50. In some rare cases the option was supported in 7.50 but documented only in the read.me.

18.1 Major New Features

Interfunction Value Tracking -- The powerful inter-statement value tracking introduced in earlier versions of PC-lint/FlexeLint has been extended to cross function boundaries. Functions called with specific values are later processed with these values used to initialize parameters. To take full advantage of interfunction tracking, a multi-pass operation has been introduced. [See Section 9.2.2 Interfunction Value Tracking.](#)

A number of interesting new warnings and gotchas have been implemented. We now detect

- pointers to insufficiently large areas (see messages [433](#), [826](#))
- the dreaded `malloc(strlen (s+1))` (see [432](#))
- the foolish `new int(10)` (see [1556](#))
- when 64 bit pointers meet with 32 bit integer arithmetic (see [679](#), [680](#))
- when pointer parameters can be made pointers to `const` ([818](#)) and when reference parameters can be declared as `const ref` ([1764](#)).
- when a member function can be made `const` ([1762](#), [1763](#), [1962](#))
- when member pointers are copied within copy constructors and assignment operators ([1554](#) and [1555](#)).
- when a test of assignment is not surrounded by parentheses (see [820](#)).
- Operator sequences when unparenthesized can be confusing to the reader such as `A/B*C` (see [834](#)).
- Returning a temporary via a reference ([1413](#)).
- A reference initialization that can cause the dreaded capability gain ([1561](#)).
- Exception irregularities ([1559](#), [1560](#) and [1775](#)).
- Cases where the `dynamic_cast` should be used ([1774](#)) and where it is not appropriate ([1939](#)).
- When string literals are placed in harm's way via assignment to `char *` ([1776](#)).
- When white space may intervene between a backslash and a newline creating an invisible ambiguity ([434](#)).
- Strange comparisons that can't possibly succeed. See message [685](#).

New Elective Notes are always of interest and they allow you to look for

- use of 0 when NULL is intended ([910](#)).
- situations where `const` can be added to parameters, and variables, and pointers (see [952](#), [953](#), and [954](#)).
- non const non volatile variables to assist multi-threaded applications in detecting non re-entrant situations. (See [956](#)).
- prototypes missing parameter names ([955](#)).
- suspicious conditions when you suspect you have order-of-evaluation problems ([1935](#), [1936](#), [1938](#)).
- pointer subtractions which can be a problem in conditions of maximum memory ([947](#)).

New and revised options of interest:

- `-sem` enhancements include the ability to specify that return pointers from functions are freshly allocated and that pointer arguments to functions are consumed (i.e. the function takes custody of the pointer). See [Section 10.2.1 Possible Semantics](#): `custodial(i)`, `type(i)`, `malloc(exp)`, `new(exp)` and `new[](exp)`.
- User-defined `printf` and `scanf` codes (See `-printf_code` and `-scanf_code` in [Section 5.7 Other Options](#)).
- A 'poor-man's' class hierarchy for C code. (See [Section 5.5 Flag Options](#) Flag `fss`).
- Better support for defining non-standard preprocessing commands. See `-ppw_asgn` and `--ppw` in [Section 5.7 Other Options](#).
- Better support for defining and manipulating reserved words. See `-rw_asgn` and `--rw` in [Section 5.7 Other Options](#).

Other options and features include the following

- To better support intelligent editors and IDE's we produce special additional messages when a given message contains interior location information (messages [830](#) and [831](#)).
- We now require a specific form of lint comment when programmer's are falling through from one (nontrivial) case to the next. See option `-fallthrough`.
- The `_lint` predefined identifier now contains version information. For Version 8.00 it has a value of 800.
- We have added explicit support for the MISRA (Motor Industry Software Reliability Association) standard. See messages [960](#), [961](#) and `au-misra.lnt`.
- Alignment is checked (messages [958](#) and [959](#)) and can be set via a `-a` option similar to the `-s` options to set sizes. [See Section 5.3 Size and Alignment Options](#).
- According to the Standard, `new` (and `new[]`) do not return NULL when out of storage but rather throw an exception (`bad_alloc`).

- Support for Microsoft C/C++ and compatible systems include: `.dsp` processing ([See Section 3.6 .dsp processing.](#)), `#import` ([Section 13.4.6 #import](#)) and pragmas `once` and `message` (See `+pragma` in [Section 5.7 Other Options](#)).
- Support for Gnu C/C++ consists of a `-cgnu` option ([Section 5.8 Compiler Adaptation](#)), variable macro arguments and a form of 'testable' assignment (Info [820](#)) that non-Gnu users can use.
- Support for Borland's C++ Builder including truncated file names (See [Section 5.5 Flag Options](#) Flag `ftr`).
- Continued improvement in support for embedded systems compilers. See `@ support` and `_to_eol` in [Section 5.8.2 Compiler Codes](#) and `asm` functions in [Section 5.8.6 In-line assembly code](#).

18.2 New Error Inhibition Options

[See Section 5.2 Error Inhibition Options.](#)

```
-etemplate( # [,#] ... ) inhibits
+etemplate( # [,#] ... ) re-enables
    error message(s) # while expanding templates.

-e{ # [,#]...}    will inhibit message number(s) # for the next statement or declaration.
--e{ # [,#]...}  will inhibit message number(s) # for the entire braced region in which it is placed.
```

18.3 New Verbosity Options

[See Section 5.4 Verbosity Options.](#)

```
-v...c    Issues a verbosity message whenever a unique function call is recorded.

-v...d    Indicates whether modules are taken as C or C++.

-v...h-   If the 'h' is followed immediately by a '-' then the hierarchy will be compressed producing the
           same tree in half the lines.

-v...t    Indicates when a template is being instantiated.

-v...w    Indicates when a function is being walked with specific arguments.
```

18.4 New Flag Options

[See Section 5.5 Flag Options.](#)

fdl	has been enhanced to allow for pointer difference to be long long .
flm	Lock Message format
fnn	new can return Null
frl	Reference Location information
fsp	SPecific function call processing
fss	Sub-struct allows C struct's to be related as in inheritance.
ftg	turns off TRiGraph processing.
ptr	TRuncate flag to support Borland include's.

18.5 New Message Presentation Options

[See Section 5.6 Message Presentation Options.](#)

-append(# , <i>string</i>)	can be used to append a trailing message (<i>string</i>) onto an existing error message for error #.
-------------------------------------	--

18.6 Additional Other Options

[See Section 5.7 Other Options.](#)

-background	PC-lint/FlexeLint will run at a reduced priority under WindowsNT.
-fallthrough	The flow of control is falling through from one case switch to another.
-headerwarn(<i>filename</i> [, <i>filename</i>] ...)	Can be used to alert the user that a header file is being used.
-ident1(<i>string</i>)	This option allows the user to define a one-character identifier.
+libdir	Enhanced to support wild cards
+libh	Enhanced to support wild cards
+linebuf	This option doubles the size of the line buffer, which is initially 600 bytes.
-passes(<i>k</i> [, <i>Options1</i>] [, <i>Options2</i>])	Allows multiple passes over the set of modules provided as input.
-ppw_asgn(<i>word1</i>, <i>word2</i>)	Assigns the preprocessor semantics associated with <i>word2</i> to <i>word1</i> and

activates *word1*.

`--ppw(word1 [, word2] ...)` Removes preconceived notions as to what the preprocessor word(s) mean.

`+pragma(identifier, once)` This option allows the programmer to establish an arbitrary identifier as an indicator that the header is to be included just once.

`+pragma(identifier, message)`

This option allows the programmer to establish an arbitrary *identifier* as a pragma that will produce a message to standard out.

`-pragma(identifier)` This option will remove the pragma whose name is *identifier*.

`-printf_code(Code [, Type])`

This option allows user-defined `printf` codes.

`--rw(word1 [, word2] ...)` Removes preconceived notions as to what the potential reserved word(s) mean and gives it a default meaning (to ignore it).

`-rw_asgn(word1, word2)` This option supports special non-standard conventions provided by some given compiler.

`-scanf_code(Code [, Type])`

This option allows user-defined `scanf` codes.

`-sem` This option has been enhanced to support a number.

`--u` This option is like `-u` except that any module at a lower `.int` level is ignored.

18.7 Compiler Adaptation

[See Section 5.8 Compiler Adaptation.](#)

`-cgnu` This option is used for the Gnu compiler.

[See Section 5.8.3 Customization Facilities.](#)

`_to_eol` This adds to the suite of special keywords that enable PC-lint/FlexeLint to pass over unusual non-standard constructions. [See Section 5.8.3 Customization Facilities.](#)

`@` PC-lint/FlexeLint now supports the `@` feature appearing in many embedded systems compilers. [See Section 5.8.3 Customization Facilities.](#)

18.8 New Pre-Processor Directives

See [Section 13.4.6 #import](#) and [Section 13.4.7 #unassert](#).

`#import` supports the Microsoft preprocessor directive of the same name.

`#unassert` preprocessor directive undoes the effects of `#assert`.

18.9 New Messages

See [Section 17.1 C Syntax Errors](#).

```
37  Value of enumerator 'Symbol' inconsistent (conflicts with Location)
90  Option 'String' is only appropriate within a lint comment
91  Line exceeds Integer characters (use +linebuf)
92  Negative array dimension or bit field length (Integer)
93  New-line is not permitted within string arguments to macros
145 Type expected before operator, void assumed
147 sizeof takes just one argument
154 Could not evaluate type 'String', int assumed
155 Ignoring {...} sequence within an expression, 0 assumed
```

See [Section 17.4 C Warning Messages](#).

```
430 Character '@', taken to specify variable location, is not standard C/C++
432 Suspicious argument to malloc
433 Allocated area not large enough for pointer
434 White space ignored between back-slash and new-line
435 integral constant 'String' has precision Integer, use +fll to enable long long
436 Preprocessor directive in invocation of macro 'Symbol' at Location
437 Passing struct 'Symbol' to ellipsis
675 No prior semantics associated with 'Name' in option 'String'
678 Member 'Symbol' field length (Integer) too small for enum precision (Integer)
679 Suspicious Truncation in arithmetic expression combining with pointer
680 Suspicious Truncation in arithmetic expression converted to pointer
681 Loop is not entered
682 sizeof applied to a parameter 'Symbol' whose type is a sized array
683 function 'Symbol' #define'd, semantics may be lost
684 Passing address of auto variable 'Symbol' into caller space
685 Relational operator 'String' always evaluates to 'String'
```

See Section 17.5 C Informational Messages.

801 Use of goto is deprecated
818 Pointer parameter '*Symbol*' (*Location*) could be declared ptr to const
820 Boolean test of a parenthesized assignment
821 Right hand side of assignment not parenthesized
825 control flows into case/default without -fallthrough comment
826 Suspicious pointer-to-pointer conversion (area too small)
827 Loop not reachable
828 Semantics of function '*Symbol*' copied to function '*Symbol*'
830 Location cited in prior message
831 Reference cited in prior message
832 Parameter '*Symbol*' not explicitly declared, int assumed
833 Symbol '*Symbol*' is typed differently (*String*) in another module, compare with *Location*.
834 Operator '*Name*' followed by operator '*Name*' is confusing. Use parentheses.

See Section 17.6 C Elective Notes.

909 Implicit conversion from *Type* to bool
910 Implicit conversion (*Context*) from 0 to pointer
947 Subtract operator applied to pointers
951 Pointer to incomplete type '*Symbol*' employed in operation
952 Parameter '*Symbol*' (*Location*) could be declared const
953 Variable '*Symbol*' (*Location*) could be declared as const
954 Pointer variable '*Symbol*' (*Location*) could be declared as pointing to a const
955 Parameter name missing from prototype for function '*Symbol*'
956 Non const, non volatile static or external variable '*Symbol*'
957 Function '*Symbol*' defined without a prototype in scope
958 Padding of *Integer* byte(s) is required to align member on *Integer* byte boundary
959 struct size (*Integer* bytes) is not an even multiple of the maximum member alignment (*Integer* bytes)
960 Violates MISRA Required Rule *Name*, *String*
961 Violates MISRA Advisory Rule *Name*, *String*
970 Use of modifier or type '*Name*' outside of a typedef
971 Use of 'char' without 'signed' or 'unsigned'

See Section 17.7 C++ Syntax Errors.

1077 Could not evaluate default template parameter '*String*'
1078 class '*Symbol*' should not have itself as a base class
1079 Could not find '>' or ',' to terminate template parameter at *Location*
1080 Definition for class '*Name*' is not in scope

See Section 17.8 C++ Warning Messages.

1412 Reference member '*Symbol*' is not initialized
 1413 function '*Symbol*' is returning a temporary via a reference
 1553 struct '*Symbol*' declared as extern "C" contains C++ substructure '*Symbol*'
 (*Location*)
 1554 Direct pointer copy of member '*Symbol*' within copy constructor: '*Symbol*'
 1555 Direct pointer copy of member '*Symbol*' within copy assignment operator: '*Sym-*
bol'
 1556 'new Type(integer)' is suspicious
 1557 const member '*Symbol*' is not initialized
 1559 Uncaught exception '*Name*' may be thrown in destructor '*Symbol*'
 1560 Uncaught exception '*Name*' not on throw-list for function '*Symbol*'
 1561 Reference initialization causes loss of const/volatile integrity (*Context*)

See Section 17.9 C++ Informational Messages.

1760 Redundant template '*Symbol*' defined identically at *Location*
 1762 Member function '*Symbol*' could be made const
 1763 Member function '*Symbol*' marked as const indirectly modifies class
 1764 Reference parameter could be declared const reference
 1768 Virtual function '*Symbol*' has an access (*String*) different from the access
 (*String*) in the base class (*String*)
 1769 Member or base class '*Symbol*' has no constructor
 1770 function '*Symbol*' defined without function '*String*'
 1771 function '*Symbol*' replaces global function
 1772 Assignment operator '*Symbol*' is not returning *this
 1773 Attempt to cast away const (or volatile)
 1774 Could use dynamic_cast to downcast ptr to polymorphic type '*Symbol*'
 1775 catch block does not catch any declared exception
 1776 Converting string literals to char * is deprecated (*Context*)

See Section 17.10 C++ Elective Notes.

1935 Dynamic initialization for class object '*Symbol1*' (references '*Symbol2*')
 1936 Dynamic initialization for variable '*Symbol1*' (references '*Symbol2*')
 1937 Static variable '*Symbol*' has a destructor.
 1938 constructor '*Symbol*' accesses global data.
 1939 Down cast detected.
 1961 virtual member function '*Symbol*' could be made const.
 1962 Non-const member function '*Symbol*' contains a deep modification.

19. BIBLIOGRAPHY

Bibliography for C

- [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
<http://www.dkuug.dk/JTC1/SC22/WG14>.
- [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, <http://www.misra.org.uk>

Bibliography for C++

- [10] *ISO/IEC 14882:1998*
International Standard – Programming Languages – C++.
- [11] Ellis, M. H. 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] Koeing, 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] Koeing, 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.

Appendix A

Option Summary

Error Inhibition Options

(See [Section 5.2 Error Inhibition Options.](#))

(- inhibits and + enables messages)

<i>Option</i>	<i>Description Default</i>
<code>-/+e#</code>	message number(s) #
<code>-e(#)</code>	message # for the next expression
<code>--e(#)</code>	message # for the current expression
<code>-/+eai</code>	argument sub-Integer on
<code>-/+ean</code>	arguments differing nominally on
<code>-/+eas</code>	arguments same size (different type) on
<code>-/+eau</code>	arguments differing signed-unsigned on
<code>-/+efile(#,file)</code>	control message by number, filename
<code>-/+efunc(#,Symbol)</code>	message # issued from within functions
<code>-/+elib(#)</code>	control message(s) in library headers
<code>-/+elibsym(#)</code>	for a library symbol
<code>-/+emacro(#,Symbol)</code>	within a macro
<code>-emacro((#),Symbol)</code>	within expression macro
<code>--emacro((#),Symbol)</code>	for entire expression in which macro is embedded
<code>-/+epn</code>	pointer to nominally different types on
<code>-/+epnc</code>	pointer to chars nominally different on
<code>-/+epp</code>	pointer vs. pointer (same size) on
<code>-/+eps</code>	pointer to same-size on
<code>-/+epu</code>	pointer to unsigned (vs. signed) on
<code>-/+epuc</code>	pointer to signed/unsigned chars on
<code>-/+esym(#,Symbol)</code>	control message by number, symbol
<code>-/+etd(TypeDiff)</code>	ignore certain type differences
<code>-/+etemplate(# [,#] ...)</code>	error messages(s) while expanding templates
<code>!e#</code>	one-line error suppression
<code>-wlevel</code>	set warning level
<code>-wlib(level)</code>	set warning level for library

Size Options

(See Section 5.3 Size and Alignment Options.)

<i>Option</i>	<i>Description</i>
-sb#	number of bits in a byte
-sc#	sizeof(char) becomes #
-slc#	sizeof(long char) becomes #
-ss#	sizeof(short) becomes #
-si#	sizeof(int) becomes #
-sl#	sizeof(long) becomes #
-sf#	sizeof(float) becomes #
-sd#	sizeof(double) becomes #
-sld#	sizeof(long double) becomes #
-sll#	sizeof(long long) becomes #
-sp#	size of all pointers becomes #
-spD#	size of Data pointer becomes #
-spP#	size of Program pointer becomes #
-spN#	size of near pointer becomes #
-spF#	size of far pointer becomes #
-spND#	size of near Data pointer becomes #
-spNP#	size of near Program pointer becomes #
-spFD#	size of far Data pointer becomes #
-spFP#	size of far Program pointer becomes #
-smp#	size of Member Pointer becomes #
-smpD#	size of Member Data Pointers
-smpP#	size of Member Program Pointer
-smpFP#	size of far Member Program Pointers
-smpNP	size of near Member Program Pointers
-sw#	size of wchar_t becomes #

Verbosity Options

(See Section 5.4 Verbosity Options.)

(- output to `stdout`, + to `stderr` and `stdout`)

<i>Option</i>	<i>Description</i>	<i>Default</i>
<code>-v</code>	Turn off all work in progress messages	
<code>-/+vm</code>	Module names only	<code>-vm</code>
<code>-/+vf</code>	File and module names only	
<code>-/+vn</code>	Every <i>n</i> lines with file and module names	
<code>-/+vc...</code>	prints a message when a function with unique arguments is called	
<code>-/+vd...</code>	appends a language dialect identifier	
<code>-/+vi...</code>	Indirect filenames	
<code>-/+vo...</code>	output Options	
<code>-/+vs...</code>	append a Storage report	
<code>-/+vt...</code>	prints a message each time a template is to be instantiated	
<code>-/+vw...</code>	specific walk	
<code>-/+vh...</code>	dump the type Hierarchy	
<code>-/+vh-...</code>	hierarchy will be compressed	
<code>-/+v#...</code>	tag filenames with an id number	

Flag Options

(See Section 5.5 Flag Options.)

(+ sets,- resets, ++ increments and -- decrements flag)

<code>fab</code>	ABbreviated structure flag	<code>OFF</code>
<code>fai</code>	Arguments pointed to get Initialized flag	<code>ON</code>
<code>fan</code>	ANonymous union flag	<code>OFF</code>
<code>fas</code>	Anonymous struct flag	<code>OFF</code>
<code>fba</code>	Addressability flag	<code>OFF</code>
<code>fbc</code>	Binary Constant flag	<code>OFF</code>
<code>fbo</code>	Boolean flag	<code>ON</code>
<code>fbu</code>	Bit fields are Unsigned flag	<code>OFF</code>
<code>fcd</code>	CDecl is significant flag	<code>OFF</code>
<code>fce</code>	Continue-on-Error flag	<code>OFF</code>
<code>fcp</code>	C++ flag	<code>OFF</code>
<code>fct</code>	Create Tag flag	<code>OFF</code>
<code>fcu</code>	Char-is-Unsigned flag	<code>OFF</code>
<code>fdc</code>	Distinguish-plain-Char flag	<code>ON</code>
<code>fdh</code>	dot-h flag	<code>OFF</code>
<code>fdi</code>	Directory of Including file flag	<code>varies</code>
<code>fdl</code>	pointer-Difference-is-Long flag	<code>OFF</code>
<code>fdr</code>	Deduce-Return-mode flag	<code>OFF</code>
<code>feb</code>	allow Enumerations as Bit fields flag	<code>ON</code>

fem	Early Modifiers flag	OFF
fep	Extended Pre-processor variable flag	OFF
ffb	for loop creates separate Block flag	ON
ffd	promote Floats to Doubles flag	OFF
fff	Fold Filenames flag	ON
ffn	Full (file) Name flag	OFF
ffo	Flush Output files flag	ON
fhd	strong-Hierarchy-Down flag	ON
fhg	Hierarchy Graphics flag	varies
fhs	Hierarchy-of-Strong-types flag	ON
fhx	Hierarchy-of-strong-indeXes flag	ON
fie	Integer-model-for-Enum flag	OFF
fil	Indentation-check-on-Labels flag	OFF
fim	Include-Multiple flag	ON
fiq	Ignore-default-Qualifier flag	OFF
fis	Integral-constants-are-Signed flag	OFF
fkp	K&R Preprocessor flag	OFF
flb	LiBrary flag	OFF
flc	long-char flag	OFF
flf	process Library-Function flag	OFF
fll	long-long flag	OFF
flm	Lock Message format flag	OFF
fln	#line directives flag	ON
fmd	Multiple Definitions flag	OFF
fna	Allow operator new[] flag	ON
fnc	Nested Comments flag	OFF
fnt	Nested Tag flag	ON
fod	Output-Declared-objects flag	OFF
fol	Output-Library-objects flag	OFF
fpa	PAuse flag	OFF
fpc	Pointer Casts retain lvalue flag	OFF
fpm	Precision is limited to the Max. args. flag	OFF
fpn	Pointer-parameter-may-be-NULL flag	OFF
fps	Parameters within Strings flag	OFF
frb	Read Binary flag	OFF
frl	Reference Location information in Messages flag	ON
fsa	Structure-Assignment flag	ON
fsc	String constants are const char flag	OFF
fsh	SHared reading flag	OFF
fsp	SPecific flag	ON
fss	Sub-Struct flag	ON
fsu	String Unsigned flag	OFF
ftf	raw-Template-Function flag	OFF
ftg	TriGraph flag	ON
ftr	TRuncate flag	OFF
ful	Unsigned Long flag	ON
fva	Variable Arguments flag	OFF
fvo	VOid data type flag	ON
fvr	Varying-Return-mode flag	OFF

fwc	wchar_t is built-in flag	OFF
fwu	wchar_t is Unsigned flag	OFF
fxa	eXact-Array flag	OFF
fxc	eXact-Char flag	OFF
fxf	eXact-Float flag	OFF
fxs	eXact-Short flag	OFF
fzl	siZeof-is-Long flag	OFF
fzu	siZeof-is-Unsigned flag	ON

Message Presentation Options (See Section 5.6 Message Presentation Options.)

<i>Option</i>	<i>Description</i>	<i>Default</i>
-h[s][F][f][a][b][r][mn][m][m/M/][I]N	controls the height of messages s - blank line between messages F - always include a filename f - out-of-sequence file information r - repeat source line mn - no macro display m - undo effect of mn m/M/ - assigns new prefix <i>M</i> for macro display I - error location mark a <i>I</i> appears above the source line b <i>I</i> appears below the source line N - number of lines in error message	-ha_3
-width(Width,Indent)	controls width of messages <i>Width</i> - Width of output device <i>Indent</i> - Indent continued lines	-width(79,4)
-format=	message format for msg. height<=3 "(%f %l %)%t %n: %m"	
-format4a=	format of first line for msg. height == 4 "(%File %f, Line %l\n%)"	
-format4b=	format of second line for msg. height == 4 "(% %)%t %n: %m"	
-append(errno,string)	appends a trailing message	
-format_specific	prologue to a Specific Walk error message	

Other Options

(See Section 5.7 Other Options.)

<i>Option</i>	<i>Description</i>
-A	requests only ANSI keywords
-/+b	suppresses or diverts the Banner line
-background	under Windows-NT will have the effect of PC-lint in the background
-ccode	specifies a particular Compiler
-/+cpp(extension)	add (+) or remove (-) a C++ extension
-dname[=value]	Define preprocessor variables
+dname=value	like -d but locks in a definition
-dname[=value][; ...]	like -d but allows multiple definitions
+ext(extension[,extension]...)	specify list for default extensions
-fallthrough	the flow of control is falling through from one case switch to another.
-father(Parent,Child[,Child]...)	adds strict relationship to strong type hierarchy
-function(f0, f1...)	assign special attributes
-header(filename)	automatically read header
-headerwarn(filename [filename]...)	Can be used alert the user that a header file is being used
-idirectory	#include file directory
-ident(String)	augment identifier characters
-ident1(String)	define a one-character identifier
-idlen(n,options)	check for identifier clashes
-incvar(name)	<i>name</i> replaces INCLUDE environment variable
-index(flags,ixtype,sitype,...)	specifies strong index
+libclass(...)	default library headers (default: foreign,angle)
-/+libdir(directory,...)	library header directory
-/+libh(file,...)	library header
-library	current or next module is a library
-limit(n)	specify limit on number of messages
+linebuf	doubles the size of the line buffer
-/+lnt(extension[,extension]...)	add or remove an Indirect File extension
-lobbase(filename)	establish a lob base file
+macros	increases the size of macro storage.
-maxopen(n)	maximum number of simultaneously open files
-mD	far Data, near program
-mL	far data and program
-mP	far program, near data
-mS	near data and program (Default)
-odopts[width] (filename)	Output Declarations (prototypes)
+odopts[width] (filename)	Append Declarations
<i>opts are:</i>	
	f only functions
	i internal (static) functions also
	s structs also
	<i>width</i> set output width
-oe(filename)	redirects output for stderr to the named file

<code>+oe(filename)</code>	Append output for <code>stderr</code> to the named file
<code>-ol(filename)</code>	Output Library
<code>-oo[(filename)]</code>	Output Object module (default: <code>Name.lob</code>)
<code>-os(filename)</code>	Output Standard out to file
<code>+os(filename)</code>	Append Standard out to file
<code>-p[(width)]</code>	run just the Preprocessor
<code>-parent(Parent, Child...)</code>	adds to strong type hierarchy
<code>-passes(k [,Options1 [,Options2]])</code>	controls the number of passes
<code>-ppw(word,...)</code>	removes PreProcessor Word(s)
<code>+ppw(word,...)</code>	adds PreProcessor Word(s)
<code>-ppw_asgn(word1, word2)</code>	assigns the preprocessor semantics
<code>--ppw(word1 [, word2] ...)</code>	removes preconceived notions as what the processor word(s) mean
<code>+pragma(name,action)</code>	allows mimicing some <code>pragma</code> systems
<code>+pragma(identifier,once)</code>	establish an arbitrary identifier as an indicator that the header is to be used once
<code>+pragma(identifier,message)</code>	establish identifier as a pragma that will produce a message to standard out
<code>-printf(n,name,...)</code>	<code>printf</code> -like functions
<code>-printf_code(Code [, Type])</code>	allows user-defined <code>printf</code> codes
<code>-restore</code>	resets error inhibition state
<code>-rw(word,...)</code>	removes Reserved Word(s)
<code>+rw(word,...)</code>	adds Reserved Word(s)
<code>--rw(word1 [, word2] ...)</code>	removes preconceived notions as to what reserved word(s) mean
<code>-rw_asgn(word1, word2)</code>	assigns keyword semantics
<code>-save</code>	saves error inhibition state
<code>-scanf(n,name,...)</code>	<code>scanf</code> -like functions
<code>-scanf_code(Code [, Type])</code>	allows user-defined <code>scanf</code> codes.
<code>-sem(name [,sem] ...)</code>	user-defined function semantics
<code>-size(flags,amount)</code>	helps to flag large aggregates
<code>-specific(Options1 [, Options2])</code>	allows the user to specify options before a Specific Walk
<code>-specific_climit(n)</code>	specifies a Call limit
<code>-specific_wlimit(n)</code>	specifies a Walk limit
<code>-strong(flags,name,...)</code>	specifies strong types
<code>-t#</code>	set Tab size to # (default = 8)
<code>-u</code>	unit checkout (suppresses inter-module messages)
<code>--u</code>	any module at a lower <code>.1nt</code> level is ignored
<code>-unreachable</code>	a point in a program is unreachable
<code>-uname</code>	Undefines <code>name</code>
<code>--uName</code>	like <code>-u</code> but inhibits subsequent defines
<code>-wLevel</code>	set Warning level
<code>-wlib(Level)</code>	set Warning level for messages within library.
<code>-wprintf(N, name1 [, name2] ...)</code>	wide character variant of <code>-printf</code>
<code>-wscanf(N, name1 [, name2] ...)</code>	wide character variant of <code>-scanf</code>

<code>-zero</code>	sets exit code to 0
<code>-zero(#)</code>	same unless a message number < #
<code>-\$</code>	\$ is an identifier character

Compiler Dependency Options

(See [Section 5.8.3 Customization Facilities.](#))

<i>Option</i>	<i>Description</i>
<code>-a#predicate</code>	Assert the # predator (Unix)
<code>-#dname=Replacement</code>	<code>#include</code> Define
<code>-dname()=Replacement</code>	Define function-like macro
<code>-overload(X)</code>	sets flags affecting overload resolution
<code>-plus(char)</code>	define an alternate option character for '+'
<code>-template(X)</code>	fine-tune template processing

Compiler Dependency Keywords

(See [Section 5.8.3 Customization Facilities.](#))

<i>Option</i>	<i>Description</i>
<code>_bit</code>	one bit wide
<code>_gobble</code>	causes the next token to be gobbled
<code>_ignore_init</code>	causes initializer to be ignored
<code>_to_brackets</code>	ignore next expression
<code>_to_semi</code>	ignore up to and including semi-colon
<code>_to_eol</code>	see page 107.

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 then 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 -l10 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`.

Symbols

!0 197	^@ 90
!e# 261	__1167 88
!e# option 29, 327	__asm 89
# stringize operator 54	__assert 152, 157
#... 60	__based 89
#asm 95	__BIGCODE 88
#assert 184	__BIGDATA 88
#c_include 185	__cdecl 89
##d option 93, 335	__CDECL__ 89
#define 25, 65, 202, 229	__COMPACT__ 89
#define limit 216	__cplusplus 182
#elif not K&R 229	__DATE__ 182
#endasm 95	__ECO 88
#endif 228	__except 89
#error 216	__export 89
#ident 74, 187	__far 89
#import 186	__farlb 89
#include 10, 46, 182, 183, 202, 205	__fastcall 89
#include file directory 332	__FILE__ 182
#include_next 186	__fortran 89
#line 184	__FUNC__ 87
#line directives flag 330	__handle 89
##pragma option 75	__huge 89
#unassert 185, 186	__import 89
-\$ option 84, 334	__inline 89
% operator 210	__interrupt 89
%ENVIR_VAR% 21	__iob 89
'^u ^-u' 9	__LARGE__ 89
) 250	__LDATA 87
*ms 87	__LINE__ 182
+/-elib 279	__loadds 89
+/-Page, pr option 337	__MEDIUM__ 89
##d option 332	__MSDOS__ 89
.c extension 20	__near 89
.cpp extension 20	__pascal 89
.dsp processing 13	__saveregs 89
.lnt file 20	__seg 89
.lob file 73, 106, 107, 108, 192	__segment 89
.txt 18	__segment 89
/* LIBRARY */ 23	__self 89
/*lint 29, 65, 79, 178, 193	__SMALL__ 89
/*lint comment 23, 24	__ss 89
// comment 64	__STDC__ 182
//lint 29	__stdcall 89
//lint comment 23	__syscall 89
	__TIME__ 182

__try 89
 __TURBOC__ 89
 _align(k) 93
 ARCHITECTURE 89
 _asm 89
 _based 89
 _bit 91, 335
 _C86_BIG 88
 _cdecl 89
 _cs 89
 _ds 89
 _es 89
 _export 89
 _far 89
 _farlb 89
 _fastcall 89
 _fortran 89
 _gobble 91, 335
 HIGHC 88
 _huge 89
 _ignore_init 91, 335
 INTELC32 89
 _interrupt 89
 _LARGE 89
 _lint 82, 87, 105, 171, 182
 _loadds 89
 _near 89
 _pascal 89
 _pragma 92
 _saveregs 89
 _seg 89
 _segment 89
 _segname 89
 _self 89
 _ss 89
 _stdio_defs_included 88
 _syscall 89
 TIMESTAMP 89
 _to_brackets 79, 92, 335
 _to_eol 92, 320
 _to_semi 92, 335
 Numerics
 32-bit code 18
 386 89
 A
 -A option 64, 89, 90, 180, 182, 332

-a# option 92, 335
 ABbreviated structure flag 329
 abort 152, 156
 abstract type 215
 access specifiers 280
 actual 89
 additional calls 139
 align 89
 allocation flag 163
 allow Enumerations as Bit fields flag 330
 ambiguous reference 239
 angle bracket header 70, 98
 Anonymous struct flag 329
 anonymous union 243, 287
 ANonymous union flag 329
 ANSI 1, 64, 89, 111, 236, 251
 ANSI C 4, 50, 168, 175, 182, 196, 231, 253
 ANSI compiler 84, 102
 ANSI header 70, 99
 ansi header 99
 ANSI keywords 332
 ANSI limit 261
 ANSI preprocessing 184
 ANSI X3J11 4
 ANSI X3J16 4
 -append option 63, 331
 Archimedes C 85
 architectures, native 38
 arg. 'this' 202
 arg. no. 202
 ARGSUSED 178
 argument 201
 argument lists 57
 argument semantics 158, 160, 166
 argument type mismatch 30
 Arguments pointed to get Initialized flag 329
 arguments, variable 179
 arithmetic 201
 array 58
 Array facility, New 52
 array matching, exact 174
 array name 228
 array size 228
 asctime 152
 ASM 75
 asm 89

- asm preprocessor directive 75
- asm.l 105
- assembly language 75, 104, 182
- assert() 136
- assert.h 99
- assignment 202
- assignment to strong type 114
- atof 152
- atoi 152
- atol 152
- au-....lnt 8
- auto 235
- auto data 80
- autodecrement 167, 202, 208, 229
- AUTOEXEC.BAT 15
- autoincrement 167, 202, 208, 229
- automatic variables 133
- available memory, exceeded 216
- Aztec C, Manx 85, 87

B

- b option 64, 332
- b#, pr option 336
- background option 64, 319
- background option 332
- banner line 64, 332
- base class 287, 288
- base-specifier 280
- basic 203
- batch file 113
- batch procedures 22
- bibliography 324
- Binary Constant flag 329
- binary operators 161
- Bit Addressability flag 329
- bit field 227, 228
- bit field sizes 199
- bit fields 47, 115
- Bit fields are Unsigned flag 329
- bit operators 208
- bit-wise operator 243
- blanks within options 24
- bool 45
- Boolean 201, 209, 250
- Boolean AND 167
- Boolean argument to function 251
- Boolean argument to relational 225

- Boolean condition 136
- Boolean flag 329
- Boolean OR 167
- Boolean test of assignment 250
- Boolean type-checking 115
- Boolean, constant value 225
- Boolean, strongly 201
- Borland C 89
- Borland C/C++ 85, 87
- Borland C/C++ OS2 85
- Borland IDE 13
- Borland/Turbo C/C++ 85
- braces 168
- bsearch 152
- BSO Tasking 85
- byte size 38

C

- C comment 308
- C Set 2 88
- C++ Elective Notes 308
- C++ extents 11, 64
- C++ flag 329
- C++ Informational Messages 296
- C++ Syntax Errors 278
- C++ Warning Messages 286
- c, pr option 336
- CadUL 85
- Call limit 81
- Call, Specific 41
- calloc 152, 156
- canonical function 195
- Cargill, Tom 325
- case 169
- cast 126, 225, 229, 232, 236
- categories, message 201
- caztec option 87
- cbc option 87
- cc86 option 88
- ccode option 87, 332
- cdecl 89
- CDecl is significant flag 329
- cdesm option 88
- cdl option 88
- cec option 88
- cgnu option 88
- CH_TIME 89

- Char 202
- char matching, exact 174
- char types 46
- character constant 254
- Char-is-Unsigned flag 329
- chc option 88
- check, indentation 168
- checking, format 168
- checking, maximum 198
- child type 122, 124
- choose.exe 17
- cht option 88
- cibm32 option 88
- cicb option 89
- class members 280
- class specifier, storage 213
- class, static 238
- clc option 89
- clc6 option 89
- clearerr 152
- cmisc option 78, 86, 87, 89
- cmisc option 44
- cmwc option 89
- co-....lnt 6, 8, 17
- co.lnt 6, 17, 86
- co-arch.lnt 85
- co-aztec.lnt 85
- co-bc.lnt 85
- co-bc4.lnt 85
- co-bc5.lnt 85
- co-bcos2.lnt 85
- co-bs166.lnt 85
- co-c86.lnt 85
- co-c86p.lnt 85
- co-cb.lnt 85
- co-cc386.lnt 85
- co-code.lnt 87
- Code Builder, Intel 89
- Code Works QC88 85
- code, 32-bit 18
- code, exit 111
- codes, compiler 87
- co-desm.lnt 85
- co-diab.lnt 85
- co-dl.lnt 85
- co-ec.lnt 85
- co-frank.lnt 85
- co-gnu.lnt 85
- co-hc.lnt 85
- co-hp.lnt 85
- co-hpacc.lnt 85
- co-ht.lnt 85
- co-ibm32.lnt 85
- co-ibmc2.lnt 85
- co-icb.lnt 85
- co-in.lnt 85
- co-in386.lnt 85
- co-in96.lnt 85
- co-kc166.lnt 85
- co-kc51.lnt 85
- co-lc2.lnt 85
- co-lc33.lnt 85
- co-lc6.lnt 85
- co-mc386.lnt 85
- co-mc86.lnt 86
- comma operator 167, 225
- command line 12, 20
- comment, lint 178
- comment, nested 234
- commenting out code 52
- comments in indirect files 21
- comments, options within 23
- compact model 180
- compatible enum 243
- compiler codes 87
- compiler dependencies 84
- compiler options file 84, 86
- compiler symbols 69
- Computer Innovations 85
- Computer Innovations C86 85, 88
- co-misc.lnt 78, 86
- co-misc20.lnt 86
- co-misc40.lnt 86
- co-misc50.lnt 86
- co-mswin.lnt 86
- conditional evaluation 167
- conditional include 185
- conditional, preprocessor 52
- CONFIG.BAT 16
- config.exe 6, 7
- CONFIG.SYS 15
- Configuration 7

- configurations, switching among 7
- conflicting definials 176
- const 212, 235, 253, 299
- const checking 170
- constant type 199
- constant value Boolean 225
- constant, character 254
- constructor 297
- constructor, copy 299
- Context 202
- Continue-on-Error flag 329
- controlling program 21
- controlling the messages 11
- conversion by prototype 270
- conversion type name 279
- co-powrc.lnt 86
- copy constructor 284, 299
- co-qc88.lnt 85
- co-sc.lnt 86
- co-sc7.lnt 86
- Cosmic C 85
- co-sun.lnt 86
- co-tc.lnt 86
- co-ti320.lnt 86
- co-tsc.lnt 86
- co-unix.lnt 86
- count 203
- co-vage.lnt 86
- co-wc16.lnt 86
- co-wc32.lnt 86
- co-wcos2.lnt 86
- co-wh.lnt 86
- co-ztc.lnt 86
- CPM68 89
- CPM80 89
- CPM86 89
- cpp option 11, 14, 64, 66, 332
- Create Tag flag 329
- critical 89
- CR-LF 55
- cross-compilers 91
- csc option 89
- ctc option 89
- ctime 152
- ctsc option 90
- ctype.h 99

- custodial 159, 165
- custodial(i) 158
- customization facilities 90
- cwc option 90
- cwh option 90

D

- D option 65, 332
- d option 24, 65, 78, 82, 91, 93, 332, 335
- d, pr option 336
- d3, pr option 336
- dangerous 421 220
- data objects 199
- data pointer 39, 328
- data pointer, far 39
- data pointer, near 39
- data variable size 80
- Datalight C 85, 88
- deallocation 246
- debuggers 178
- debugging 188
- declaration 202
- declaration, function 97, 211
- declaration, local 178
- declaration, member 310
- declaration, old-style function 211
- declarations, output 129
- Deduce-Return-mode flag 329
- default 169
- default argument 281
- default flags 128
- define 332
- define preprocessor variables 65
- defined 226
- definials, conflicting 176
- definials, redundant 176
- definials, unused 176
- definials, weak 175
- definition 202, 310
- definition, function 97, 174
- definition, language 4
- definition, local 178
- delete 282, 288, 297
- derived class 288
- DeSmet C 85, 88
- destructor 279, 288
- development environment, integrated 12

- Diab Data C 85
- Directory of Including file flag 329
- directory, library 70
- Distinguish-plain-Char flag 329
- DLC 88
- dname option 65
- dname= option 65
- DOS 89
- DOS_H 89
- dot-h flag 329
- down assignments 124
- Duncan, Ray 3

E

- e 250
- e option 26, 178, 327
- e(option 26, 27
- e(#) option 327
- e(#) option 33, 327
- e) 250
- e713 option 189
- e718 option 189
- e737 option 189
- e746 option 189
- ea option 30
- eai option 30, 226, 327
- ean option 30, 199, 226, 327
- Early Modifiers flag 330
- eas option 30, 199, 226, 327
- eau option 30, 189, 226, 327
- Ecosoft Eco-C88 85, 88
- editor interface 12, 60, 63, 81
- efile option 26, 30, 202, 327
- efunc option 327
- efunc option 26, 30
- elective notes 173, 201, 268
- elib option 26, 31, 83, 327
- elibrary option 26, 31, 327
- ellipsis 179, 195, 203, 210, 230, 309
- emacro option 26, 32, 196, 327
- emacro(option 32
- EMM386 18
- end of line 55
- ENDASM 75
- enum 173, 176, 212
- enum model, integer 49
- enum model, strict 49

- enum, compatible 243
- enumeration 47, 243
- env-....lnt 6, 8, 17
- env-BC5.lnt 13
- env-bwin.lnt 13
- env-cw.lnt 13
- environment variable 15, 21, 69
- environment variable, INCLUDE 68, 183
- environment variable, LINT 20
- env-pwb.lnt 13
- env-si.lnt 13
- env-sled.lnt 13
- env-tide.lnt 13
- env-vc.lnt 13
- env-vc2.lnt 13
- env-vc4.lnt 13
- env-vc5.lnt 13
- EOL 228
- ep option 34
- epn option 34, 199, 209, 327
- epnc option 35, 327
- epp option 35, 209
- epp otion 327
- eps option 35, 199, 209, 327
- epu option 34, 209, 327
- epuc option 35, 327
- errno.h 99
- error 123 195
- error indicator 61
- error inhibition by file 30
- error inhibition by symbol 35
- error inhibition in library headers 31
- error inhibition options 26, 327
- error inhibition settings 77, 79
- error messages 204
- error suppression 36
- error(_) 131
- errors, fatal 201, 216
- errors, internal 201, 216
- errors, syntax 201
- escape sequence 236
- esym option 26, 31, 35, 193, 203, 327
- etd option 37, 179, 203, 327
- etd(option 129
- etemplate option 26
- etemplate(option 37

- evaluation, order of 167, 171, 231
- exact array matching 174
- exact char matching 174
- exact float matching 174
- exact parameter matching 173
- exact short matching 174
- eXact-Array flag 331
- eXact-Char flag 331
- eXact-Float flag 331
- eXact-Short flag 331
- example of a policy 188
- exceeded available memory 216
- exclusive index type 118
- EXECSQL 198
- exit 137, 152, 156
- exit code 21, 84, 111, 334
- exit function 244
- exit(_) 158
- exp 160
- explicit initializer 251
- ext(option 66, 332
- Extended Pre-processor 47
- Extended Pre-processor variable flag 330
- extension, .c 20
- extents, C++ 64
- extern 225
- external names 178
- external symbols 69
- extracting strong type 114

F

- fab flag option 43, 329
- fai flag option 43, 329
- fallthrough option 66, 319, 332
- false 45
- FAMILY 89
- fan 243
- fan flag option 43, 329
- far 89
- far data pointer 39, 72, 328
- far pointer 39, 72, 180, 181, 237, 328
- far program pointer 39, 328
- fas flag option 44, 329
- fatal errors 201, 216
- father option 67, 122, 124
- father(option 332
- fba flag option 44, 329
- fbc flag option 44, 329
- fbo flag option 45, 329
- fbu flag option 45, 329
- fcd flag option 45, 329
- fce flag option 45, 216, 329
- fclose 152, 157
- fcf 11, 66
- fcf flag option 45, 329
- fct flag option 45, 329
- fcu flag option 45, 329
- fdc flag option 46, 329
- fdh flag option 46, 329
- fdi flag option 46, 183, 329
- fdl flag option 46, 219, 329
- fdr 254
- fdr flag option 47, 227, 329
- feb flag option 47, 330
- fem flag option 47, 330
- feof 152
- fep flag option 47, 330
- ferror 152
- ffb flag option 48, 330
- ffd flag option 48, 330
- fff flag option 48, 330
- fflush 152
- fflush(_) 48
- ffn flag option 48, 330
- ffo flag option 48, 330
- fgetc 152
- fgetpos 152
- fgets 152, 156, 162
- fhd flag option 49, 330
- fhg flag option 49, 126, 330
- fhs flag option 49, 122, 124, 330
- fhx flag option 49, 330
- fie flag option 49, 330
- field sizes, bit 199
- field, bit 227, 228
- fil flag option 49, 170, 330
- file identification numbers 42
- file limit 217
- file scope 281
- file, .lob 73
- file, error inhibition by 30
- file, make 84
- filea.cpp 9, 17

- fileb.cpp 9, 17
- FileName 202
- fim flag option 50, 330
- fiq flag option 50, 330
- fis flag option 50, 330
- fkp flag option 50, 226, 229, 330
- flag 330
- flag, #line directives 330
- flag, ABbreviated structure 329
- flag, allow Enumerations as Bit fields 330
- flag, Anonymous struct 329
- flag, ANonymous union 329
- flag, Arguments pointed to get Initialized 329
- flag, Binary Constant 329
- flag, Bit Addressability 329
- flag, Bit fields are Unsigned 329
- flag, Boolean 329
- flag, C++ 329
- flag, CDecl is significant 329
- flag, Char-is-Unsigned 329
- flag, Continue-on-Error 329
- flag, Create Tag 329
- flag, Deduce-Return-mode 329
- flag, Directory of Including file 329
- flag, Distinguish-plain-Char 329
- flag, dot-h 329
- flag, Early Modifiers 330
- flag, eXact-Array 331
- flag, eXact-Char 331
- flag, eXact-Float 331
- flag, eXact-Short 331
- flag, Extended Pre-processor variable 330
- flag, Flush Output files 330
- flag, Fold Filenames 330
- flag, for loop creates separate Block 330
- flag, Full (file) Name 330
- flag, Hierarchy Graphics 330
- flag, Hierarchy-of-strong-indeXes 330
- flag, Hierarchy-of-Strong-types 330
- flag, Ignore-default-Qualifier 330
- flag, Include-Multiple 330
- flag, Indentation-check-on-Labels 330
- flag, Integer-model-for-Enum 330
- flag, Integral-constants-are-Signed 330
- flag, K&R Preprocessor 330
- flag, LiBrary 330
- flag, Lock Message format 330
- flag, long-char 330
- flag, long-long 330
- flag, Multiple Definitions 330
- flag, Nested Comments 330
- flag, Nested Tag 330
- flag, Output-Declared-objects 330
- flag, Output-Library-objects 330
- flag, Parameters within Strings 330
- flag, PAuse 330
- flag, Pointer Casts retain lvalue 330
- flag, pointer-Difference-is-Long 329
- flag, Pointer-parameter-may-be-NULL 330
- flag, Precision is limited to the Max. args. 330
- flag, process Library-Function 330
- flag, promote Floats to Doubles 330
- flag, raw-Template-Function 330
- flag, Read Binary 330
- flag, Reference Location information in Mes-sages 330
- flag, SHared reading 330
- flag, siZeof-is-Long 331
- flag, siZeof-is-Unsigned 331
- flag, SPecific 330
- flag, String constants are Const char 330
- flag, String Unsigned 330
- flag, strong-Hierarchy-Down 330
- flag, Structure-Assignment 330
- flag, Sub-Struct 330
- flag, TriGraph 330
- flag, TRuncate 330
- flag, Unsigned Long 330
- flag, Variable Arguments 57, 330
- flag, Varying-Return-mode 331
- flag, VOid data type 330
- flag, wchar_t is built-in 331
- flag, wchar_t is Unsigned 331
- flb flag option 50, 330
- flc flag option 51, 330
- flf flag option 51, 330
- fll flag option 39, 51, 330
- flm flag option 51, 330
- fln flag option 51, 330
- float matching, exact 174
- float.h 99
- Flush Output files flag 330

- fmd flag option 52, 205, 330
- fna flag option 52, 330
- fnc flag option 52, 234, 330
- fnn flag option 52
- fnt flag option 53, 330
- fod flag option 53, 112, 330
- fol flag option 53, 112, 330
- Fold Filenames flag 330
- fopen 152
- for loop 48
- for loop creates separate Block flag 330
- for loops 130
- for(250
- for(e 250
- foreign header 98
- foreign header 70
- format 230, 231, 238
- format checking 76, 79, 168, 199, 230
- format, Message 51
- format, message 62
- format= option 62, 331
- format_specific option 63, 143, 331
- format4a= option 331
- format4b= option 331
- fortran 89
- fpa flag option 53, 330
- fpc flag option 53, 208, 330
- fpm flag option 54, 330
- fpn flag option 54, 330
- fprintf 152, 230
- fprintf option 57
- fps 236
- fps flag option 54, 330
- fputc 152
- fputs 152
- Franklin C 85
- frb flag option 55, 330
- fread 152, 156, 164
- free 152, 156
- free() 131
- freopen 152
- frexp 153
- friend declaration 282
- frl flag option 55, 330
- fsa flag option 55, 208, 330
- fsc flag option 55, 330

- fscanf 153, 230
- fseek 153
- fsetpos 153
- fsh flag option 56, 330
- fsp flag option 56, 143, 330
- fss flag option 56, 330
- fstream.h 99
- fsu flag option 56, 330
- ftell 153
- ftf flag option 56, 330
- ftg flag option 57, 330
- ftt flag option 57, 330
- ful flag option 57, 330
- Full (file) Name flag 330
- full signature 36
- function declaration 97, 211
- function definition 97, 174
- function mimicry 67, 148
- function names 156
- function option 67, 131, 137, 148, 164, 332
- function pointer 39
- function() option 157
- function, canonical 195
- function-wide semantics 164, 166
- fva flag option 57, 172, 179, 226, 330
- fvo flag option 58, 226, 227, 330
- fvr flag option 58, 227, 331
- fwc flag option 40, 58, 331
- fwrite 153, 156
- fwu flag option 40, 58, 331
- fxa flag option 58, 174, 331
- fxc flag option 59, 174, 175, 226, 331
- fxf flag option 59, 174, 175, 331
- fxs flag option 59, 174, 175, 226, 331
- fzl flag option 59, 218, 331
- fzu flag option 59, 218, 331

G

- General Processing 141
- generation, header 171
- generation, prototype 72, 102, 171
- generic compiler 86
- getc 153
- getenv 153
- gets 153, 157, 220
- global struct 257
- global template 304

gmtime 153

GNU 85

H

-h option 60, 331

-h, pr option 337

Harbison & Steele 4

header 176, 183

header generation 171

-header option 67, 82, 332

header, angle bracket 70, 98

header, ANSI 70, 99

header, ansi 99

header, -foreign 98

header, foreign 70

header, library 70, 97, 111

header, removable from 177

header, standard 102

header, unused 176

header-directory 100

header-name 100

height, message 60, 61

help 20

help screen 72, 197

Hewlett Packard 85

hierarchy 42

Hierarchy Graphics flag 330

hierarchy tree, strong type 42, 123, 125

Hierarchy-of-strong-indexes flag 330

Hierarchy-of-Strong-types flag 330

High C/C++, MetaWare 47, 85, 88

himem.sys 18

hints on strong typing 126

Hi-Tech C 85, 88

huge 89, 180

I

-i 183

-i option 7, 50, 68, 99, 183, 193, 332

-i#, pr option 337

I8086 88, 89

i8086 88

I8086D 88, 89

I8086L 88, 89

I8086P 88, 89

I8086S 88, 89

IBM 32 bit 88

IBM C/2 compiler 85

IBM Visual Age 86

IDE, Borland 13

-ident option 68, 332

-ident1(option 68, 319, 332

identification numbers, file 42

identifier 161

identifier characters 68, 84, 94

identifier length 69, 238

identifier, undefine 82

-idlen 238

-idlen option 69, 332

if statements 130

ignore initialization 91

Ignore-default-Qualifier flag 330

in Headers, Suppressing Messages 26

INCLUDE 69

include directories, multiple 50

INCLUDE environment variable 68, 69, 183

include path 68

include, conditional 185

Include-Multiple flag 330

incomplete 203

incremental linting 107

-incvar(option 69, 332

indentation check 168, 227, 251

indentation checking 81

indentation, label 49, 170

Indentation-check-on-Labels flag 330

indented # 229

-index option 69, 114, 118, 119, 128, 332

index type, exclusive 118

indexes, strong 49

indirect file 21, 23, 41, 71

indirect file, lint 12

indirect files, comments in 21

indirectly referenced pointer 128

Informational category 310

informational messages 176, 188, 201, 249

inhibition options, error 26

inhibition settings, error 77, 79

initialization 202

initialization tracking 130

initialization, ignore 91

initialization, static 179

initialization, variable 251

inline 89, 299

- in-line assembly 95
- in-progress messages 9, 40
- install program 15
- install.log 6
- int/enum 204
- Integer 202
- integer 161
- Integer-model-for-Enum flag 330
- integral 202
- integral arguments 158, 160
- Integral-constants-are-Signed flag 330
- integrated development environment 12
- Intel 80x86 180
- Intel C 85
- Intel Code Builder 85, 89
- Intel iC-386 85
- interface, editor 12, 60, 63, 81
- intermediate type 119
- inter-module messages 81
- internal errors 201, 216
- internal functions 172
- International Standards Organization 4
- interrupt 79, 89
- iostream.h 99
- ISO 4
- ixtype 118, 128
- ixtype option 69

J

- joining strong type 114

K

- K&R 1, 4, 226
- K&R Preprocessor flag 330
- Keil C 166 85
- Keil C 51 85
- keywords 86
- keywords, Microsoft 78, 90
- keywords, preprocessor 187
- keywords, register 90
- Kind 203

L

- l#, pr option 337
- label indentation 49, 170
- language definition 4
- large data model 72
- large memory model 72
- large model 180

- large program model 72
- LARGE_CASE 88
- laser printer 337
- LATTICE 89
- Lattice C 85, 89
- LC60 89
- LCOPY.BAT 15
- leading tabs 168
- left, shift 189
- length, identifier 69, 238
- level, warning 11, 82, 201
- LF 55
- lib-....h 8
- lib-....lnt 6, 8, 17
- libclass option 70, 98, 332
- libdir option 70, 98, 99, 319, 332
- libh option 67, 70, 98, 100, 319, 332
- libraries 97
- library directory 70
- library file, standard 84, 86
- LiBrary flag 330
- library flag 115
- library header 70, 97, 111
- library headers, error inhibition in 31
- library module 50, 72
- library module creation 102
- library object module 103
- library option 70, 101, 111, 178, 332
- library tag 98
- library warning level 83
- Library-Function flag, process 51
- limit messages 70
- limit(option 70, 217, 332
- limit, #define 216
- limit, macro 216
- limits.h 99, 195
- LIN.BAT 8, 15, 16
- lin.bat 7, 8, 113, 190
- line numbers 336
- linebuf option 70, 319, 332
- linker symbols 69
- lint command 20
- lint comment 23, 29, 32, 178, 196
- LINT environment variable 20
- lint indirect file 12
- lint macro 35, 196

- lint object module 53, 71, 73, 103, 104, 106, 191
- lint options 193
- lint options, Unix 178
- lint.exe 16
- lint=, set 20
- LINT_ARGS 89
- LINTLIBRARY 178
- lint-nt.exe 6, 16
- lint-os2.exe 16
- Linux 85
- lnt(option 71, 332
- lobbase(option 71, 109, 332
- local declaration 178
- local definition 178
- local structure 255
- local template 303
- locale.h 99
- localtime 153
- Location 203
- location information 55
- location unknown 175
- Lock Message format flag 330
- LONG_MIN 195
- long-char flag 330
- longjmp 153
- long-long flag 330
- LPLOGFONT 127
- LPMSG 127
- LPRECT 127
- LPSTR, 127
- LPTR 88, 89
- LPWNDCLASS 127
- LSET.BAT 8, 15
- lvalue 53, 202, 208, 209, 283

M

- M_I86 89, 90
- M_I86CM 89, 90
- M_I86LM 89, 90
- M_I86MM 89, 90
- M_I86SM 89, 90
- m68k 88
- Macintosh MPW 26
- macro 29, 69, 96, 112, 176, 196, 202, 213, 236, 255, 256
- macro display 60

- macro limit 216
- macro parameters 54
- macro size 71
- macro, lint 35, 196
- macros option 71, 332
- macros, options within 23
- make file 22, 84, 107, 110, 191
- malloc 153, 156, 163
- Manx Aztec C 85, 87
- Mark Williams C 89
- matching, exact parameter 173
- math.h 99
- max 195
- MAX, 18
- maximum checking 198
- maxopen(option 71, 332
- mbstowcs 153
- mbtowc 153
- mD option 72, 332
- medium model 72, 180
- member 202
- member declaration 310
- member function 164
- member pointer 39
- memchr 153, 156
- memcmp 153, 156
- memcpy 153, 156
- memmove 153
- memory leak 221, 222, 246
- memory model 93, 233
- memory model, large 72
- memory requirements 1
- memory, exceeded available 216
- memset 153, 156
- message 76
- message categories 201
- message file 60
- Message format 51
- message format 62
- message height 60, 61
- message presentation options 59, 331
- message processing 60
- message suppression 83
- message suppression, one-line 29
- message tags 60
- message width 62

- Messages in Headers, Suppressing 26
- messages, error 204
- messages, informational 176, 201, 249
- messages, limit 70
- messages, suppressing 193
- messages, warning 201, 218
- meta characters 25
- MetaWare High C/C++ 47, 85, 88
- Meyers, Scott 326
- Microsoft C 89, 175, 194
- Microsoft C/C++ 13, 86
- Microsoft keywords 78, 86, 89, 90, 180
- Microsoft make 110
- Microsoft Visual C++ 86
- Microsoft Visual Studio 13
- Microsoft Windows 77, 123, 127, 240
- Microtek MCC 86
- mimicry, function 148
- min 195
- mismatch, argument type 30
- mismatch, pointer-to-pointer 34
- mixed language 104
- mktime 153
- mL option 72, 332
- ml option 50
- model, large data 72
- model, large memory 72
- model, large program 72
- model, medium 72
- model, memory 233
- modf 153
- module 202
- module creation, library 102
- module, library 72
- module, library object 103
- module, lint object 73
- module1.cpp 17
- module2.cpp 17
- module3.cpp 17
- module4.cpp 17
- mP option 72, 332
- MPU8086 87
- MPW, Macintosh 26
- mS option 72, 332
- MSDOS 89
- msg.exe 201

- msg.txt 6, 17, 201
- multidimensional arrays 119
- multi-pass 138
- Multiple Definitions flag 330
- multiple include directories 50
- multiple passes 143
- Murray, Robert 325

N

- VARARGS 179
- n, pr option 337
- native architectures 38
- natural type hierarchy 122, 123
- near 89
- near data pointer 39, 328
- near pointer 39, 72, 180, 181, 328
- near program pointer 39, 328
- nested comment 234
- Nested Comments flag 330
- Nested Tag flag 53, 330
- new 163, 280, 282, 288, 297, 300
- New Array facility 52
- NO_EXT_KEYS 89, 90
- noalign 89
- nominal 203
- nominal type-checking 179
- non-ANSI 53, 64, 111, 236
- non-ANSI compiler 102
- non-ANSI compilers 84
- non-ANSI extensions 180
- non-arithmetic 209
- non-K&R 53
- non-scalar 209
- nopad 89
- NOSTRICT 179
- notes, elective 173, 201
- NOTREACHED 179
- NULL 89
- NULL not defined 194
- NULL pointer 43, 135
- null pointer 158, 162

O

- o option 216
- object module, library 103
- object module, lint 106, 191
- octal 213
- od option 72, 102, 107, 129, 171, 172, 173,

332
 -odf option 173
 -odi option 172
 -ods option 102, 173
 -oe option 72, 333
 -oe(option 197
 -offsetof option 33
 -ol option 72, 333
 old-style function declaration 211
 once 75
 one-bit type 91
 one-line message suppression 29
 -oo option 53, 73, 103, 106, 192, 333
 operator 279
 operator delete 93, 153
 operator new 52, 153, 156, 282
 option display 25
 option settings 197
 option summary 327
 option, -!e# 29, 327
 option, -#d 93, 335
 option, -#pragma 75
 option, -\$ 84, 334
 option, ++d 332
 option, -A 64, 89, 90, 180, 182, 332
 option, -a# 92, 335
 option, -append 63, 331
 option, -b 64, 332
 option, -background 64, 319
 option, -caztec 87
 option, -cbc 87
 option, -cc86 88
 option, -ccode 87, 332
 option, -cdesm 88
 option, -cdl 88
 option, -cec 88
 option, -cgnu 88
 option, -chc 88
 option, -cht 88
 option, -cibm32 88
 option, -cicb 89
 option, -clc 89
 option, -clc6 89
 option, -cmisc 78, 86, 87, 89
 option, -cmwc 89
 option, -cpp 11, 14, 64, 66, 332
 option, -csc 89
 option, -ctc 89
 option, -ctsc 90
 option, -cwc 90
 option, -cwh 90
 option, -D 65, 332
 option, -d 24, 65, 78, 82, 91, 93, 332, 335
 option, -dname 65
 option, -dname= 65
 option, -e 26, 178, 327
 option, -e(26, 27
 option, --e(#) 327
 option, -e(#) 33, 327
 option, -e713 189
 option, -e718 189
 option, -e737 189
 option, -e746 189
 option, -ea 30
 option, -eai 30, 226, 327
 option, -ean 30, 199, 226, 327
 option, -eas 30, 199, 226, 327
 option, -eau 30, 189, 226, 327
 option, -efile 26, 30, 202, 327
 option, --efunc 327
 option, -efunc 26, 30
 option, -elib 26, 31, 83, 327
 option, -elibsymb 26, 31, 327
 option, -emacro 26, 32, 196, 327
 option, -emacro(32
 option, -ep 34
 option, -epn 34, 199, 209, 327
 option, -epnc 35, 327
 option, -epp 35, 209
 option, -eps 35, 199, 209, 327
 option, -epu 34, 209, 327
 option, -epuc 35, 327
 option, -esym 26, 31, 35, 193, 203, 327
 option, -etd 37, 179, 203, 327
 option, -etd(129
 option, -etemplate 26
 option, -etemplate(37
 option, -ext(66, 332
 option, fab flag 43, 329
 option, fai flag 43, 329
 option, -fallthrough 66, 319, 332
 option, fan flag 43, 329

- option, fas flag 44, 329
- option, -father 67, 122, 124
- option, -father(332
- option, fba flag 44, 329
- option, fbc flag 44, 329
- option, fbo flag 45, 329
- option, fbu flag 45, 329
- option, fcd flag 45, 329
- option, fce flag 45, 216, 329
- option, fcp flag 45, 329
- option, fct flag 45, 329
- option, fcu flag 45, 329
- option, fdc flag 46, 329
- option, fdh flag 46, 329
- option, fdi flag 46, 183, 329
- option, fdl flag 46, 219, 329
- option, fdr flag 47, 227, 329
- option, feb flag 47, 330
- option, fem flag 47, 330
- option, fep flag 47, 330
- option, ffb flag 48, 330
- option, ffd flag 48, 330
- option, fff flag 48, 330
- option, ffn flag 48, 330
- option, ffo flag 48, 330
- option, fhd flag 49, 330
- option, fhg flag 49, 126, 330
- option, fhs flag 49, 122, 124, 330
- option, fhx flag 49, 330
- option, fie flag 49, 330
- option, fil flag 49, 170, 330
- option, fim flag 50, 330
- option, fiq flag 50, 330
- option, fis flag 50, 330
- option, fkp flag 50, 226, 229, 330
- option, flb flag 50, 330
- option, flc flag 51, 330
- option, flf flag 51, 330
- option, fll flag 39, 51, 330
- option, flm flag 51, 330
- option, fln flag 51, 330
- option, fmd flag 52, 205, 330
- option, fna flag 52, 330
- option, fnc flag 52, 234, 330
- option, fnn flag 52
- option, fnt flag 53, 330

- option, fod flag 53, 112, 330
- option, fol flag 53, 112, 330
- option, -format= 62, 331
- option, -format_specific 63, 143, 331
- option, -format4a= 331
- option, -format4b= 331
- option, fpa flag 53, 330
- option, fpc flag 53, 208, 330
- option, fpm flag 54, 330
- option, fpn flag 54, 330
- option, -fprintf 57
- option, fps flag 54, 330
- option, frb flag 55, 330
- option, frl flag 55, 330
- option, fsa flag 55, 208, 330
- option, fsc flag 55, 330
- option, fsh flag 56, 330
- option, fsp flag 56, 143, 330
- option, fss flag 56, 330
- option, fsu flag 56, 330
- option, ftf flag 56, 330
- option, ftg flag 57, 330
- option, ftr flag 57, 330
- option, ful flag 57, 330
- option, -function 67, 131, 137, 148, 164, 332
- option, -function() 157
- option, fva flag 57, 172, 179, 226, 330
- option, fvo flag 58, 226, 227, 330
- option, fvr flag 58, 227, 331
- option, fwc flag 40, 58, 331
- option, fwu flag 40, 58, 331
- option, fxa flag 58, 174, 331
- option, fxc flag 59, 174, 175, 226, 331
- option, fxf flag 59, 174, 175, 331
- option, fxs flag 59, 174, 175, 226, 331
- option, fzl flag 59, 218, 331
- option, fzu flag 59, 218, 331
- option, -h 60, 331
- option, -header 67, 82, 332
- option, -i 50, 68, 99, 183, 193, 332
- option, -ident 68, 332
- option, -ident1(68, 319, 332
- option, -idlen 69, 332
- option, -incvar(69, 332
- option, -index 69, 114, 118, 119, 128, 332
- option, ixtype 69

option, -libclass 70, 98, 332
 option, -libdir 70, 98, 99, 319, 332
 option, -libh 67, 70, 98, 100, 319, 332
 option, -library 70, 101, 111, 178, 332
 option, -limit(70, 217, 332
 option, -linebuf 70, 319, 332
 option, -lnt(71, 332
 option, -lobbase(71, 109, 332
 option, -macros 71, 332
 option, -maxopen(71, 332
 option, -mD 72, 332
 option, -mL 72, 332
 option, -ml 50
 option, -mP 72, 332
 option, -mS 72, 332
 option, -o 216
 option, -od 72, 102, 107, 129, 171, 172, 173, 332
 option, -odf 173
 option, -odi 172
 option, -ods 102, 173
 option, -oe 72, 333
 option, -oe(197
 option, -offsetof 33
 option, -ol 72, 333
 option, -oo 53, 73, 103, 106, 192, 333
 option, -os 73, 110, 111, 333
 option, -overload(93, 335
 option, -p 33, 73, 333
 option, -parent 67, 74, 123, 124
 option, -passes(74, 143, 319, 333
 option, -plus(94, 335
 option, -ppw 74, 95, 185, 187, 206, 333
 option, -ppw(74, 320, 333
 option, -ppw_asgn(74, 319, 333
 option, -pragma 75, 333
 option, -pragma(76, 320, 333
 option, -printf 57, 76, 79, 148, 168, 333
 option, -printf_code 77, 80
 option, -printf_code(77, 320, 333
 option, -restore 27, 77, 79, 333
 option, -rw 78, 91, 333
 option, -rw(78, 91, 92, 198, 320, 333
 option, -rw(*ms) 78, 87, 180
 option, -rw_asgn(78, 320, 333
 option, -save 27, 32, 77, 79, 333
 option, -sb 38, 228, 328
 option, -sc 38, 328
 option, -scanf 76, 79, 148, 168, 333
 option, -scanf_code 77, 80
 option, -scanf_code(79, 320, 333
 option, -sd 39, 328
 option, -sem 320
 option, -sem(_) 80, 157, 221, 333
 option, -sf 39, 328
 option, -si 38, 199, 328
 option, sitype 69
 option, -size(333
 option, -size(_) 80
 option, -sizeof(328
 option, -sl 39, 199, 328
 option, -slc 38
 option, -sld 39, 328
 option, -sll 39, 51, 328
 option, -smp 39
 option, -smp# 328
 option, -smpD# 39, 328
 option, -smpFP# 39, 181, 328
 option, -smpNP# 39, 181, 328
 option, -smpP# 39, 328
 option, -sp 39, 328
 option, -spD 39, 328
 option, -specific(81, 145, 333
 option, -specific_climit(81, 145, 333
 option, -specific_retry(146
 option, -specific_wlimit(81, 146, 333
 option, -spF 39, 181, 328
 option, -spFD 39, 181, 328
 option, -spFP 39, 181, 328
 option, -spN 39, 181, 328
 option, -spND 39, 181, 328
 option, -spNP 39, 181
 option, -spP 39, 328
 option, -ss 38, 328
 option, -strong 69, 81, 113, 114, 115, 128, 129, 214, 333
 option, -sw# 40, 51, 58, 328
 option, -t# 81, 168, 227, 333
 option, -template(94, 335
 option, -u 11, 14, 81, 82, 106, 333
 option, -uname 82
 option, -unreachable 81, 179, 333

- option, unreachable 131
- option, -v 40, 318, 329
- option, -v# 42
- option, -vc 41, 147, 329
- option, -vd 41, 318, 329
- option, -vf 41, 98, 329
- option, -vh 42, 329
- option, -vh- 318, 329
- option, -vhm 125
- option, -vi 41, 329
- option, -vm 41, 329
- option, -vn 41
- option, -vo 41, 329
- option, -voif 20, 197
- option, -vs 41, 329
- option, -vt 41, 318, 329
- option, -vw 41, 147, 318, 329
- option, -w 82, 83, 188, 327, 333
- option, -w2 82
- option, -wchar_t 40, 51
- option, -width 61, 62, 331
- option, -wlib(83, 327, 333
- option, -wprintf 84
- option, -wprintf(333
- option, -wscanf 84
- option, -wscanf(334
- option, -wsprintf 77
- option, -Za 89, 90, 182
- option, -zero 21, 84, 106, 111, 334
- option, -zero(334
- options 7, 12, 23
- options and meta characters 25
- options file, compiler 84, 86
- options within comments 23
- options within macros 23
- options, order of 193
- options, output 41
- options, size 38
- options.lnt 8, 12, 15, 21, 189, 190
- order of evaluation 167, 171, 231
- order of options 193
- origin 203
- os option 73, 110, 111, 333
- OS2 89
- option, -epp 327
- out-of-bounds pointer 135

- output declarations 129
- output object module 73
- output options 41
- output standard error 72
- output standard out 73
- output structures 72
- Output-Declared-objects flag 330
- Output-Library-objects flag 330
- overflow, stack 205
- overload(option 93, 335
- overloaded function 46, 93, 151, 280, 299

P

- p option 33, 73, 333
- pad 89
- parallel arrays 119
- parameter 202
- parameter matching, exact 173
- Parameters within Strings flag 330
- parent option 67, 74, 123, 124
- parent type 122, 124
- pascal 89
- passes(option 74, 143, 319, 333
- patches 13
- path, include 68
- path, search 183
- PAuse flag 330
- pc-lint.pdf^ 6
- perror 154
- Phar Lap 17, 18
- plus(option 94, 335
- pointer arguments 158, 161
- Pointer Casts retain lvalue flag 330
- pointer size 328
- pointer, data 39
- pointer, far 39, 72
- pointer, far data 39, 72
- pointer, far program 39
- pointer, function 39
- pointer, member 39
- pointer, near 39, 72
- pointer, near data 39
- pointer, near program 39
- pointer, program 39
- pointer-Difference-is-Long flag 329
- Pointer-parameter-may-be-NULL flag 330
- pointer-to-pointer mismatch 34

- policy, example of a 188
- Possible Semantics 158
- Power C 86
- ppw option 74, 95, 185, 187, 206, 333
- ppw(option 74, 320, 333
- ppw_asgn(option 74, 319, 333
- pr 336
- pr options 336
- pr.exe 6, 17, 336
- pragma option 75, 333
- pragma(option 76, 320, 333
- pragma(option 75, 333
- precautions, prototype 173
- precision 204
- Precision is limited to the Max. args. flag 330
- pre-defined symbols 182
- preprocessing, ANSI 184
- preprocessor 73, 104, 182, 251, 333
- preprocessor command words 74
- preprocessor conditional 52
- preprocessor directive 185
- preprocessor directive, asm 75
- preprocessor keywords 187
- preprocessor statements 95
- preprocessor symbols 69, 87, 182
- preprocessor variables 332
- preprocessor variables, define 65
- preprocessor, K&R 50
- presentation options, message 59
- printf 154, 168, 230
- printf option 57, 76, 79, 148, 168, 333
- printf_code option 77, 80
- printf_code(option 77, 320, 333
- private 89
- private member 283
- problems, common 193
- process Library-Function flag 51, 330
- processing, message 60
- program pointer 39, 328
- program pointer, far 39
- program pointer, near 39
- program scope 281
- Programmer's Work Bench 13
- Programs, Test 8
- promote Floats to Doubles flag 330
- promotion 195, 204

- promotion, type 173, 175, 179, 196, 200
- protected member 283
- prototype 57, 101, 102, 107, 111, 174, 189, 211, 226, 255
- prototype generation 72, 102, 171
- prototype precautions 173
- prototype, conversion by 270
- prototypes in scope 102
- prototypes, typedef types in 173
- ptrdiff_t 219
- ptrs to different types 204
- ptrs to incompatible types 204
- public member 283
- pure specifier 279
- putc 154
- puts 154

Q

- QEMM 18
- qsort 154
- qualification 204

R

- r_no 158
- r_null 158
- raw-Template-Function flag 330
- Read Binary flag 330
- readme.txt 6, 17
- realloc 154
- redirect standard error 72
- redirect standard out 73
- redirection 40, 42, 73, 111
- redundant definials 176
- Reference Location information in Messages flag 330
- referenced pointer, indirectly 128
- register keywords 90
- removable from header 177
- remove 154
- rename 154
- repeat source line 60
- reserved word 78
- reserved words 78, 86
- restore option 27, 77, 79, 333
- return 202
- return mode 47, 58
- Return Semantics 164
- Return semantics 158, 162, 166

- return semantics 149
- rewind 154
- Reznick, Larry 326
- right, shift 189
- Ritchie, Dennis 324
- rw option 78, 91, 333
- rw(option 78, 91, 92, 198, 320, 333
- rw(*ms) option 78, 87, 180
- rw_asgn(option 78, 320, 333
- S
- s, pr option 337
- S1-S20 156
- Saks, Dan 326
- save option 27, 32, 77, 79, 333
- sb option 38, 228, 328
- sbo option 38
- sc option 38, 328
- scalar 202
- scalar sizes 197
- scalars 38
- scalars, size of 199
- scanf 154, 168, 230
- scanf option 76, 79, 148, 168, 333
- scanf_code option 77, 80
- scanf_code(option 79, 320, 333
- scope operator 296
- sd option 39, 328
- search list 98
- search path 183
- sem option 320
- sem(_) 158
- sem(_) option 80, 157, 221, 333
- semantic checking 80, 157
- Semantic Expressions 161
- Semantic Specifications 166
- semantics 156
- semantics checking 158
- semantics, argument 158, 160
- sequence, escape 236
- SET INCLUDE= 183
- set lint= 20
- setbuf 154
- setjmp 154, 156
- setjmp.h 99
- settings, option 197
- SETUP 6

- setvbuf 154
- sf option 39, 328
- SHared reading flag 330
- shift 209, 232
- shift left 189
- shift right 189
- shift, unusual 225
- short matching, exact 174
- si option 38, 199, 328
- signal.h 99
- signature 35
- signature, full 36
- signed/unsigned 189, 204
- Simpson, Steve 310
- sitype 118, 128
- sitype option 69
- size 204
- size of scalars 199
- size options 38, 210, 328
- size(option 333
- size(_) option 80
- size, array 228
- size, byte 38
- size, macro 71
- size_t 218
- sizeof 59, 89, 199, 210, 214, 218, 226, 246
- sizeof(option 328
- sizeof(all pointers) 39
- sizeof(char) 38
- sizeof(double) 39
- sizeof(float) 39
- sizeof(int) 38
- sizeof(long char) 38
- sizeof(long double) 39
- sizeof(long) 39
- sizeof(short) 38
- siZeof-is-Long flag 331
- siZeof-is-Unsigned flag 331
- sizes, scalar 197
- sl option 39, 199, 328
- sl-....c 6, 17
- sl.c 6, 17, 101, 111
- slc option 38
- sl-c86.c 85
- sld option 39, 328
- sll option 39, 51, 328

- sl-lc2.c 85
- small model 72, 180
- smp option 39
- smp# option 328
- smpD# option 39, 328
- smpFP# option 39, 181, 328
- smpNP# option 39, 181, 328
- smpP# option 39, 328
- source line, repeat 60
- sp option 39, 328
- spD option 39, 328
- special keywords 92
- Specific Call 41
- Specific Call Portrayal 141
- Specific Calls 140
- SPecific flag 330
- Specific Walk 41, 63, 81, 138
- Specific Walks 140
- specific(option 81, 145, 333
- specific_climit(option 81, 145, 333
- specific_retry(option 146
- specific_wlimit(option 81, 146, 333
- spF option 39, 181, 328
- spFD option 39, 181, 328
- spFP option 39, 181, 328
- spN option 39, 181, 328
- spND option 39, 181, 328
- spNP 328
- spNP option 39, 181
- spNP, 328
- spP option 39, 328
- sprintf 154, 230
- SPTR 88, 89
- Spuler, David 326
- SQL commands 198
- ss option 38, 328
- sscanf 154, 230
- stack facility 121
- stack overflow 80, 205, 216
- standard error 40
- standard error, output 72
- standard header 102
- standard library file 84, 86
- standard out 40, 73
- standard out, output 73
- standard out, redirect 73
- static 178, 225, 229, 253
- static class 238
- static data 80
- static functions 172
- static initialization 179
- STD.LNT 8, 15, 16
- std.lnt 12, 21, 190
- STD_A.LNT 16
- stdarg.h 99
- stddef.h 99
- stdio.h 97, 99
- stdlib.h 99
- storage class specifier 213
- storage consumed 41
- strcat 154, 156
- strchr 154
- strcmp 154
- strcoll 154
- strcpy 154, 156
- strcsn 154
- strftime 154
- strict type-checking 179
- String 203
- String constants are Const char flag 330
- string initializer 228
- String Unsigned flag 330
- string.h 99
- strlen 154, 156
- strncat 154, 156
- strncmp 154
- strncpy 154, 156
- strong 204
- strong expressions 127
- strong indexes 49
- strong option 69, 81, 113, 114, 115, 128, 129, 214, 333
- strong table 128
- strong type hierarchy tree 42, 123, 125
- strong type, assignment to 114
- strong type, extracting 114
- strong type, joining 114
- strong type-checking 114, 126
- strong types 49, 113
- strong typing 144
- strong-Hierarchy-Down flag 330
- strongly Boolean 201

Stroustrup, Bjarne 4, 325
 strpbrk 154
 strrchr 154
 strspn 155
 strstr 155
 strstream.h 99
 strtod 155
 strtok 155
 strtol 155
 strtoul 155
 struct 173, 176
 struct declaration, untagged 287
 struct, untagged 216
 Structure-Assignment flag 330
 structures, output 72
 strxfrm 155
 Sub-Struct flag 330
 Sun 86
 suppressing messages 193
 Suppressing Messages in Headers 26
 -sw# option 40, 51, 58, 328
 switch statement 131
 switching among configurations 7
 Symantec C/C++ 86, 89
 Symbol 35, 203
 symbol, error inhibition by 35
 symbols, pre-defined 182
 symbols, preprocessor 87
 syntax errors 201, 204

T

 -t# 251
 -t# option 81, 168, 227, 333
 tab size 81, 333
 table, strong 128
 tabs, leading 168
 Tag flag, Nested 53
 tags, message 60
 tellme.exe 18
 template 176, 282, 283, 284, 303
 template function 56
 -template(option 94, 335
 templates 112
 ternary operator 161
 Test Programs 8
 test.h 17
 test1.lnt 17
 test1.out 17
 test2.lnt 17
 test2.out 17
 Texas Instruments C 86
 this 279, 281, 283
 throw() 293
 time identifier 93
 time.h 99
 tmpfile 155
 TopSpeed C 86, 90
 torture testing 198
 tracking, initialization 130
 trigraph 253
 TriGraph flag 330
 true 45
 TRuncate flag 330
 Turbo C 89
 Turbo/Borland C/C++ 86
 Type 203
 Type = Type 204
 type differences 37
 type hierarchies 120
 type hierarchy tree, strong 42, 123, 125
 type hierarchy, natural 122, 123
 type matching 58, 59
 type mismatch, argument 30
 type promotion 173, 175, 179, 196, 200
 Type vs. Type 204
 type(i) 159
 type, child 122, 124
 type, constant 199
 type, parent 122, 124
 type, vacuous 207
 type-checking, Boolean 115
 type-checking, nominal 179
 type-checking, strict 179
 type-checking, strong 114
 typedef 113, 118, 119, 123, 176, 255, 256
 typedef types in prototypes 173
 typedefs 112
 TypeDiff 37, 203
 types, strong 49

U

 -u option 11, 14, 81, 82, 106, 249, 251, 256, 333
 -uname option 82

- unary minus operator 225
- unary operators 161
- undefine 333
- undefine identifier 82
- ungetc 155
- uninitialized statics 179
- union 173, 176
- union assignment 55
- union initialization 249
- union, anonymous 243
- unit checkout 11, 81, 106, 249, 251, 256, 333
- Unix 26, 187
- Unix C/C++ 86
- Unix lint options 178
- Unix make 110
- unknown, location 175
- unreachable 227
- unreachable code 158
- unreachable option 81, 179, 333
- unreachable option 131
- Unsigned Long flag 330
- untagged struct 216
- untagged struct declaration 287
- unused definials 176
- unused header 176
- unusual constructs 196
- unusual shift 225
- unusual use of a Boolean 225
- unwise.exe 6
- USEASM 105

V

- v option 40, 318, 329
- v# option 42
- vacuous type 207
- value tracking 133
- value-preserving 200
- variable arguments 179
- Variable Arguments flag 57, 330
- variable initialization 251
- Varying-Return-mode flag 331
- VAX-11 C 93
- vc option 41, 147, 329
- vd option 41, 318, 329
- verbosity 9, 25, 40
- verbosity option 197
- verbosity options 98

- version 9
- vf option 41, 98, 329
- vfprintf 155
- vh option 42, 329
- vh- option 318, 329
- vhm option 125
- vi option 41, 329
- virtual function 300
- virtual member function 298
- Visual Age 88
- Visual C++, Microsoft 86
- vm option 41, 329
- vn option 41
- vo option 41, 329
- void 209, 212, 213, 214, 226
- VOid data type flag 330
- void* 283
- void/nonvoid 204
- voif option 20, 197
- volatile 171, 235
- vprintf 155
- vs option 41, 329
- vsprintf 155
- vt option 41, 318, 329
- vw option 41, 147, 318, 329

W

- w option 82, 83, 188, 327, 333
- w2 option 82
- Walk limit 81
- Walk, Specific 41
- warning level 11, 82, 201
- warning level, library 83
- warning messages 201, 218
- Watcom 16-bit C/C++ 86
- Watcom 32-bit C/C++ 86
- Watcom C 90
- wchar_t is built-in flag 331
- wchar_t is Unsigned flag 331
- wchar_t option 40, 51
- wcstombs 155
- wctomb 155
- weak definials 175
- web site 13
- while loops 130
- while(e) 250
- Whitesmith C 86, 90

- width option 61, 62, 331
- width, message 62
- wild card 20, 26, 36
- Windows 1
- Windows, Microsoft 77, 123, 127, 240
- wlib(option 83, 327, 333
- words, reserved 78
- Work Bench, Programmer's 13
- wprintf option 84
- wprintf(option 333
- wscanf option 84
- wscanf(option 334
- wsprintf option 77

X

- X3J11, ANSI 4
- X3J16, ANSI 4

Z

- z80 88
- Za option 89, 90, 182
- zero option 21, 84, 106, 111, 334
- zero(#) option 334
- Zortech C/C++ 86