Reference Manual
for

PC-lint Plus

A diagnostic facility for C and C++

Gimpel Software LLC

August 2018

Gimpel Software LLC
3207 Hogarth Lane
Collegeville, PA 19426
(610) 584-4261
www.gimpel.com

Contents

1 Introduction

1.1 Overview

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

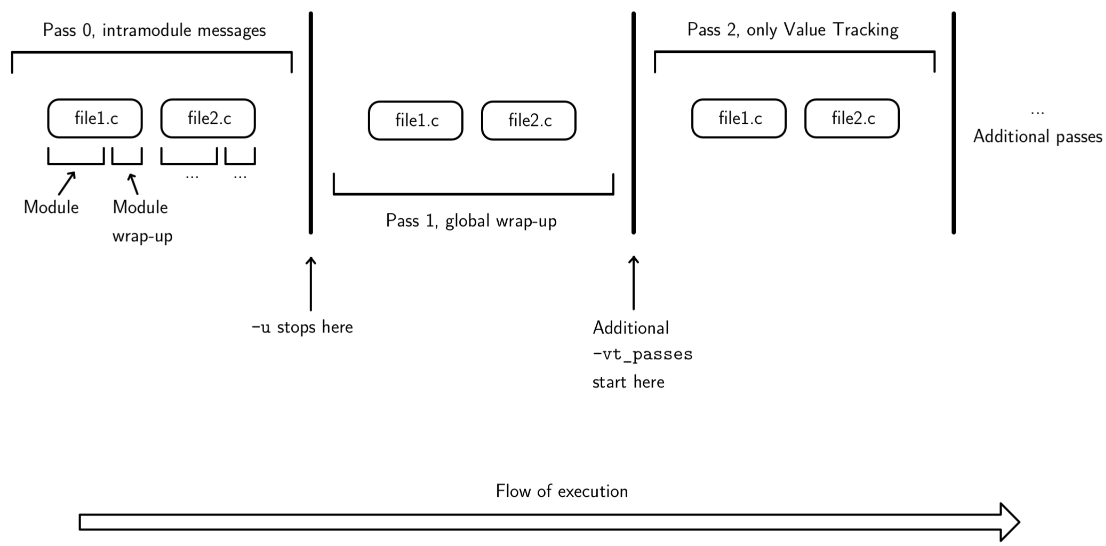
PC-lint Plus doesn't just find bugs and potential bugs, such as null pointer dereference, out of bounds access, and improper operation order, it will also point out the use of dubious constructs and substandard practices that are likely to result in buggy code or code that is difficult to reason about.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

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

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

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

1.2 Flow of Execution



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

In pass 0, most standard messages that do not require aggregate information about entire files or the entire project are emitted as PC-lint Plus examines the code. At the end of each module in pass 0, module wrap-up is performed and messages that rely on information gathered from the entirety of the module are issued (e.g. reports of unused entities not visible outside the module).

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

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

1.3 An Example

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

```c
char *report(short m, short n, char *p) {
    int result;
    char *temp;
    long nm;
    int i, k, kk;
    char name[11] = "Joe Jakeson";
    nm = n * m;
    temp = p == "" ? "null" : p;
    for (i = 0; i < m; i++) {
        k++;
        kk = i;
    }
    if (k == 1)
        result = nm;
    else if (kk > 0)
        result = 1;
    else if (kk < 0)
        result = -1;
    if (m == result)
        return temp;
    else
        return name;
}
```

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

```
--- Module:   intro_example1.c (C)
intro_example1.c 6 info 784: nul character truncated from string
    char name[11] = "Joe Jakeson";
                    ^~~~~~~~~~~~~
intro_example1.c 22 warning 604: returning address of auto variable 'name'
        return name;
               ^~~~
intro_example1.c 10 warning 530: 'k' is likely uninitialized
        k++;
        ^
intro_example1.c 5 supplemental 891: allocated here
```

```
    int i, k, kk;
           ^

intro_example1.c  8  info 779: string constant in comparison operator '=='
    temp = p == "" ? "null" : p;
             ^  ~~
```

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

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

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

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

intro_example1.c  1  note 9403: function 'report' parameter 2 has same
    unqualified type ('short') as previous parameter
char *report(short m, short n, char *p) {
             ~~~~~~~  ^
intro_example1.c  8  note 9050: dependence placed on operator precedence
    (operand of ?:)
    temp = p == "" ? "null" : p;
           ^
```

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

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

2 Installation and Configuration of PC-lint Plus

2.1 System Requirements

2.1.1 Supported Operating Systems

- Windows 7, Windows 8, Windows 10.
- OS X version 10.11 and later.
- Linux with kernel 2.6.15 or higher and glibc 2.11 or higher.

2.1.2 Hardware Requirements

Minimum Requirements

- 2 GB RAM
- x86-64 compatible CPU (x86 compatible CPU for 32-bit Windows version)

Recommended Requirements

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

2.2 Installation

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

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

2.2.1 Setting the PATH environment variable on Windows

1. Open a command or run prompt such as by pressing 'R' while holding the Windows key.
2. Type `sysdm.cpl` into the run prompt and press Enter.
3. Select the "Advanced" tab in the dialog window.
4. Click "Environment Variables".
5. In the "System Variables" section, locate the `PATH` variable and add the directory containing PC-lint Plus to the semi-colon delimited list if it does not already exist.
6. Click "OK" on this and the remaining dialog windows.

Verify the change by running echo `%PATH%` from a newly opened terminal window.
Note: You may need to have administrative privileges to alter system settings.
Note: This change will not affect already open command prompt windows.

2.2.2 Setting the PATH environment variable on Linux

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

Verify the change by running echo `$PATH` from a newly opened terminal window.
Note: This change will not affect already open terminal windows.

2.2.3 Setting the PATH environment variable on Mac OS

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

Verify the change by running echo `$PATH` from a newly opened terminal window.
Note: This change will not affect already open terminal windows.

2.3 Configuring with pclp_config

2.3.1 Overview

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

Note: `pclp_config` requires that the Python interpreter as well as the `regex` and `pyyaml` modules to be installed. See 2.3.12 Installing python and 2.3.13 Installing required modules for instructions on installing these components.

2.3.2 Introduction and Walkthrough of Automated Configuration with pclp_config

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

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

```
#include <stdio.h>

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

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

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

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

clean:
    rm hello.o hello
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 ...

// Size Options
```

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

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

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

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

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

```
hello.c  4  error 40: undeclared identifier 'SUBJECT'
    printf("Hello, %s\n", SUBJECT);
                          ^
```

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

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

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

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

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

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

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

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

and on Windows:

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

Now we can run make:

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

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

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

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

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

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

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

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

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

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

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

Both the `imposter` program and the `pclp_config` script have a number of configurable options to handle various situations. For example, some projects might not get through the build process without source files being compiled in which case `imposter` needs to be configured to call the actual compiler after it logs the options it was called with. If you are targeting an architecture that is not your compiler's default, you will need to provide the architecture options to `pclp_config` when building the compiler configuration. These details are documented in the 2.3.10 Imposter Options Reference and 2.3.9 pclp_config Options Reference sections.

2.3.3 Creating a compiler configuration for GCC or Clang

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

   ```
   which gcc
   ```

   and on Windows with the command:

   ```
   where gcc
   ```

   replacing `gcc` with the name of your compiler.

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

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

   For clang use:

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

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

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

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

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

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

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

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

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

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

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

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

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

   In Windows 10:

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

   In Windows 8.1:

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

   In Windows 8:

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

   In Windows 7:

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

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

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

   ```
   where cl
   ```

3. Run the `pclp_config.py` script:

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

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

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

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

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

   See 2.3.12 Installing Python for steps to install Python if necessary.

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

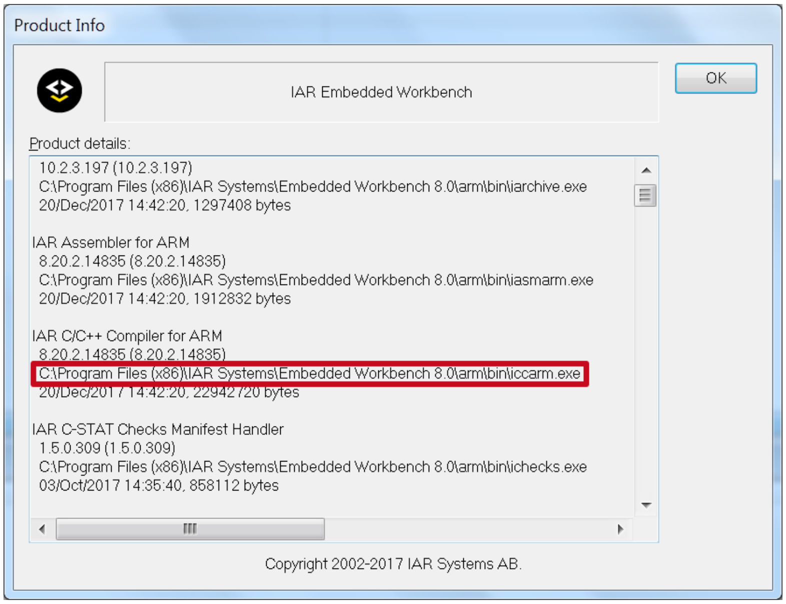
The generated configuration will be suitable for both C and C++ source files.

2.3.5 Creating a compiler configuration for IAR Embedded compilers

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

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

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



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

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

The `--compiler-options` option specifies the options passed to the IAR compiler when compiling your project, make sure to replace `...` with the options used to compile your project.

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

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

See 2.3.12 Installing Python for steps to install Python if necessary.

The generated configuration will be suitable for both C and C++ source files.
See 2.3.8 Integrating PC-lint Plus with IAR Embedded Workbench for instructions on integrating PC-lint Plus with the IAR IDE.

2.3.6 Creating a project configuration with make or cmake

1. Open a command prompt and locate the compiler used in your build process with the `which` or `where` command as shown in step #1 of 2.3.3 Creating a compiler configuration for GCC or Clang.

2. Set the `IMPOSTER_LOG` environment variable to the full path of the file that will hold compiler invocations. On most shells this command will look like:

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

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

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

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

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

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

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

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

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

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

```
export IMPOSTER_COMPILER_ARG1
```

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

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

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

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

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

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

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

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

2.3.7 Creating a project configuration with MSBuild on Windows

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

1. Follow steps #1 and #2 from Creating a compiler configuration for Microsoft C/cpp compilers to open a Developer Command Prompt and locate the `cl.exe` binary.

2. Locate the Visual Studio project or solution file for your project. Solution files have a `.sln` extension.

3. In the Developer Command Prompt, set the `IMPOSTER_LOG` environment variable to the full path where compiler invocations should be logged:

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

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

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

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

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

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

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

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

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

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

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

2.3.8 Integrating PC-lint Plus with IAR Embedded Workbench

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

1. In the Tools menus, select "Configure Tools".

2. In the configuration dialog box that opens up, click "New".

3. For "Menu Text" enter "Lint Current File", you might want to add text to indicate the chipset if you will be configuring PC-lint Plus for multiple IAR compiler.

4. For "Command", enter the location of the PC-lint Plus binary.

5. For "Argument", enter the following:

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

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

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

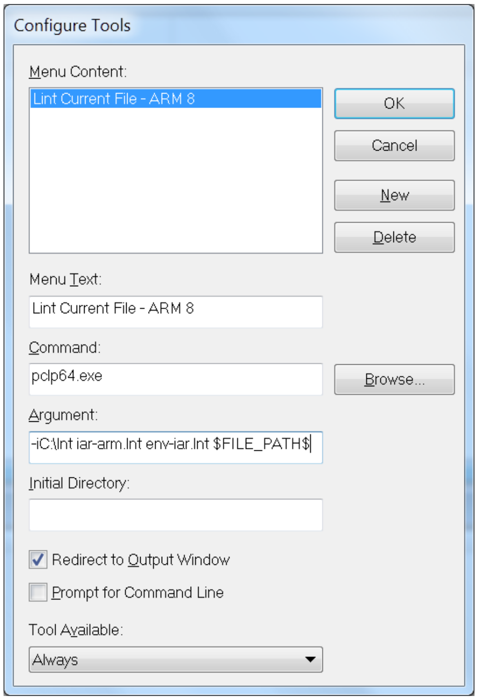
   `"-iC:\location with spaces\lint"`

6. Check the box "Redirect to Output Window".

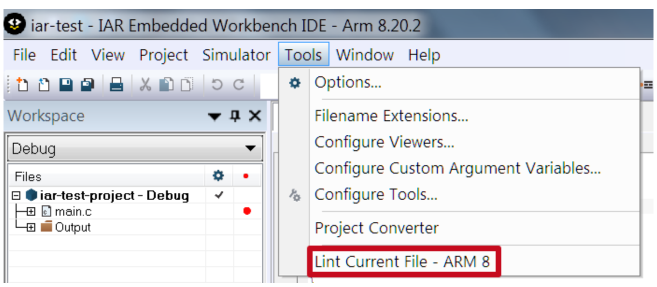
7. Select "Always" from the "Tool Available" dropdown.

8. Click on "OK".

The result should look something like:



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



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

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

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

2.3.9 pclp_config Options Reference

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

Option	Description
`--help`	Show the help screen.
`--compiler-database`	Used to specify the location of the compiler database; `compilers.yaml` is the default.
`--list-compilers`	Dump a list of compilers supported by the compiler database along with a short description of each.
`--compiler`	Used to specify the *name* of a compiler or compiler family from the list of supported compilers.
`--compiler-bin`	Used to specify the full path of the compiler binary for operations that need access to the compiler (such as the `--generate-compiler-config` and `--compiler-version` operations).
`--compiler-version`	Runs the specified compiler and attempts to extract and display version information. Useful for troubleshooting. Emits 'None' if extraction fails for any reason. Must be used with `--compiler` and `--compiler-bin`.
`--generate-compiler-config`	Instructs `pclp_config` to generate a compiler configuration for the specified compiler. Usually used with `--compiler` and `--compiler-bin`.
`--generate-project-config`	Instructs `pclp_config` to generate a project configuration using compile commands logged from running the build process with the *imposter* program in place of the compiler. Used with the `--compiler`, `--imposter-file`, and `--config-output-lnt-file` options.
`--config-output-lnt-file`	Specifies the name of the configuration file to write when generating compiler or project configurations.
`--config-output-header-file`	Specifies the name of the compiler configuration header file to generate.
`--imposter-file`	Specifies the name of the log file containing compile commands logged by the *imposter* process.
`--source-pattern`	A regular expression used to distinguish source modules from options in the compile commands log generated by the imposter process. The default value is `.*\.(c|cpp)$` and matches any string that ends with `.c` or `.cpp`.
`--compiler-options`	A space separated list of base compiler options. If you are targeting an architecture other than the compiler's default, you should include the option that specifies the target architecture when generating a compiler configuration to ensure the correct values for size options in the generated configuration. Compiler options specified with this option are applied when the compiler is invoked in either C or C++ mode.

`--compiler-c-options` Similar to `--compiler-options` but only used when invoking the compiler in C mode. Use for C-only language options such as setting the language version.

`--compiler-cpp-options` Similar to `--compiler-options` but only used when invoking the compiler in C++ mode. Use for C++-only language options such as setting the language version.

`--ignore-options` Specify compiler options in a compile commands file that should not be transformed into PC-lint Plus options. If the compiler options begin with a `-`, you will want to use the `--option=arg` form when using this option. This option can be specified multiple times to ignore multiple compiler options.

`--repl` Start a Read Eval Print Loop where compiler options are read from stdin and transformed PC-lint Plus options are printed to stdout. Requires `--compiler`. This is a debugging option.

`--scavenge-files` Specifies the list of files to attempt to extract macro names from when performing macro scavenging.

`--scavenge-dirs` Specifies the directories to recurse looking for files to extract macro information from when performing macro scavenging.

`--scavenge-pattern` A regular expression specifying files that should be examined for macro information when using `--scavenge-dirs`.

2.3.10 imposter Options Reference

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

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

`IMPOSTER_PRE_2008_PARAMS` When parsing response files, the parameters it contains are processed using the Windows command line parameter parsing rules. There is a subtle and undocumented difference between the way handling of consecutive quotes was handled prior to 2008 and after 2008. By default, the post-2008 rules are employed. If this variable is set, the pre-2008 rules are instead employed.

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

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

2.3.11 Installing pclp_config prerequisites

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

2.3.12 Installing python

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

- Windows
  Download and run the latest Python 2 release for Windows.
- Linux
  Python is probably already installed; try running `python --version` to check. If it isn't, you can use a package manager to install it:
  - `sudo apt-get install python or sudo apt-get install python2` on Ubuntu
  - `sudo yum install python` on RedHat or Fedora

  You can also install python from source:
  - Download the python sources, uncompress, and run `./configure && make altinstall`.

  See Using Python on Unix platforms for more information.
- MacOSX
  Python is installed by default on Mac OS X.

- FreeBSD Python is probably already installed; try running `python --version` to check. If it isn't, you can install it with the command `pkg_add -r python` or build it from source.

  See Using Python on Unix platforms for more information.

2.3.13 Installing required modules

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

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

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

- On Mac OS: open a terminal window and run: `sudo easy_install pip`
- On other platforms: Download the `get-pip.py` script and run it from a terminal window with `python get-pip.py`.

2.4 System Configuration

Step 1 - Configure Include Directories using the -i option

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

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

  Then, use the command:

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

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

- If the compiler provides an option to list system include paths, this option can be employed. For gcc and clang based compilers, the options `-v -xc -E /dev/null` will emit information that includes the system include search path for C headers. For C++ include paths, the corresponding options are `g++ -v -xc++ -E /dev/null`. For other compilers, consult the compiler's documentation to determine if such an option exists.

- An intentional compile-time error can be introduced in such a way to elicit a message from the standard header, the error message would presumably include the path information. For example, the following program:

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

  produces the following error when processed by clang:

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

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

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

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

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

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

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

Step 2 - Extract Predefined Macros

PC-lint Plus needs to know about compiler-defined macros since these will be used by the implementation. Many compilers provide an option to dump such predefined macros to a file. This is by far the easiest way to extract the macros if your compiler supports it. If your compiler doesn't support such an option, you'll need to utilize the macro scavenging feature described below.

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

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

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

- **Method 2 - Extract macros using the macro scavenger**

  PC-lint Plus provides Macro Scavenger utility to assist in the process of identifying and configuring compiler defined macros. See the Macro `scavenge` option for additional information.

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

Step 3 - Establish the sizes of the fundamental types

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

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

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

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

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

Step 4 - Enable compiler-specific extensions

Most compilers implement a number of extensions to the C and C++ languages, such as GCC's Case Ranges or Microsoft's Assembly blocks. Embedded compilers often support additional language extensions. If these extensions are used by projects that will be processed by PC-lint Plus, they should be enabled in the configuration. Because there is a wide range of extensions supported by various compilers, we provide starter configuration files for many popular compilers. These files are given names of the form `co-`*compiler*`.lnt` and are distributed with the product and provided on our website. Locate the compiler configuration file that most closely corresponds to the compiler you are using. For example, if you are using GCC or clang, select the `co-gcc.lnt` file, if you are using Visual Studio 2015, select the `co-vs2015.lnt` file, etc. If you do not see a configuration that matches your exact compiler, look for one that is similar, e.g. the same product family, etc. Many compilers are derivatives of popular compilers like GCC or clang. If you are using a derivative product and do not see a configuration file for your specific compiler, choose the one that is associated with the compiler that the derivative is based on. If there are no files that appear to be applicable to your compiler, choose `co-generic.lnt` and contact support if you run into issues configuring PC-lint Plus for your compiler.

Putting it all Together

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

- `lint-includes.lnt` contains the system include paths.
- `lint_cmac.h` and/or `lint_cppmac.h` contains C and C++ compiler macro definitions.
- `size-options.lnt` contains options describing the sizes of the fundamental data types.
- A compiler options file, such as `co-gcc.lnt` or `co-vs2015.lnt`

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

```
lint-includes.lnt
size-options.lnt
co-generic.lnt  // Replace with the actual compiler options file selected
```

3 The Command Line

3.1 Indirect (.lnt) files

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

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

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

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

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

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

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

`std.lnt` might contain:

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

This illustrates the nesting of indirect files.

3.2 Exit Code

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

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

3.3 Built-in version environment variables

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

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

- `LINT_PATCH_LEVEL`
  The patch level, e.g. 1.

- `LINT_VERSION`
  The full version number incorporating the previous three components, using two digits for the minor version as described in Section 12.1 Preprocessor Symbols, e.g. 1001.

- `LINT_BETA_LEVEL`
  For a beta release, a number representing the beta level, e.g. 1. For non-beta releases, this variable has the value 0.

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

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

4 Options

4.1 Rules for Specifying Options

Options begin with a plus (+) or minus (-), except as for the special !e option. Strings that start with any other character are presumed to be filename arguments. [1]

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

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

4.1.1 Options within Comments

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

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

or

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

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

4.1.2 Lint Comments inside of Macro Definitions

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

```
#define UNUSED
```

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

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

The seemingly equivalent:

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

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

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

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

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

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

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

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

4.1.3 Options on the Command Line

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

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

4.1.4 Specifying Option Arguments

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

4.1.5 Quoted Options and Arguments

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

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

will emit the message:

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

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

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

will emit the message:

```
This is a test
```

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

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

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

will emit the message:

```
,!)) )
```

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

4.1.6 Quotes in Arguments

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

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

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

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

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

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

4.1.7 Braces within Argument Lists

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

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

Prints the text:

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

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

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

4.1.8 Braces and Quotes

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

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

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

4.1.9 Brace Types in Brace-Enclosed Argument Lists

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

4.1.10 Option Display

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

4.1.11 Expansion of Environment Variables in Options

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

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

Note that because of the expansion of environment variables occur before the options are processed, a `-setenv` option will not affect following options in the same line comment. In general, we recommend that environment variables be defined at the beginning of processing and not be changed afterwards.

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

4.1.12 Escaping Special Characters

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

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

4.2 Option Reference

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

Option	Summary	Version
`?`	displays help	—
`-A`	specifies the C or C++ language version	—
`-I`	add search *directory* for `#include` directives	—
`-$`	permit `$` in identifiers	—
`-a`	set the alignment of various types	—
`-append`	append *String* to diagnostic `#` when issued	—
`+b`	redirect banner output to stdout	—
`++b`	produce banner line	—

Option	Summary	Version
-b	suppress banner output	—
-cond	conditionally execute options	1.0
+cpp	add C++ extension(s)	—
-cpp	remove C++ extension(s)	—
+d	define the preprocessor symbol *Name* resistent to change via `#define`	—
++d	define the preprocessor symbol *Name* that cannot be `#undef`'d	—
-d	define the preprocessor symbol *Name* with value *Value*	—
-deprecate	deprecates the use of *Name* within *Category*	—
-dump_message_list	dumps PC-lint Plus message list to the provided file	1.0
-dump_messages	dumps PC-lint Plus messages to the provided file in the specified format	1.0
!e	disables message # for the current line	—
-e(	inhibits message #s for the next expression	—
--e(	inhibits message #s for the entire enclosing expression	—
+e	re-enables message(s) #	—
-e	disables a message where # is a message number or pattern	—
-e{	inhibits message #s for the next statement	—
--e{	inhibits message #s for the entire enclosing braced region	—
+ecall	enables the message #s within calls to *Function*	—
-ecall	inhibits the message #s within calls to *Function*	—
+efile	enables the message #s within *File*	—
-efile	inhibits the message #s within *File*	—
+efreeze	freeze the message #s and/or warning level(s)	—
++efreeze	deep freeze the message #s and/or warning level(s)	—
-efreeze	unfreeze the message #s and/or warning level(s)	—
+efunc	enables the message #s within the body of *Function*	—
-efunc	inhibits the message #s within the body of *Function*	—
+egrep	enables the message #s when the message text matches *Regex*	1.0
-egrep	inhibits the message #s when the message text matches *Regex*	1.0
+elib	enables message #s in library code	—
-elib	disables the message #s in library code	—
+elibcall	enables message #s inside calls to library functions	—
-elibcall	inhibits message #s inside calls to library functions	—
+elibmacro	enables message #s issued for library macros	—
-elibmacro	inhibits message #s issued for library macros	—
+elibsym	enables message #s issued for library symbols	—
-elibsym	inhibits message #s issued for library symbols	—
+emacro	enables message #s within macro expansions	—
-emacro	inhibits message #s within macro expansions	—
--emacro	inhibits message #s within macro expansions	—
-env_pop	pop the current option environment	1.0
-env_push	push the current option environment	1.0
-env_restore	restore the option environment to a previously saved one	1.0
-env_save	save the current option environment with name *Name*	1.0
+estring	enables the message #s parameterized by *String*	—
-estring	inhibits the message #s parameterized by *String*	—
+esym	enables the message #s parameterized by *Symbol*	—

Option	Summary	Version
`-esym`	inhibits the message `#`s parameterized by *Symbol*	—
`+etype`	enables the message `#`s parameterized by *Type*	—
`-etype`	inhibits the message `#`s parameterized by *Type*	—
`-exitcode`	set the exit code to n	1.0
`+ext`	set the extensions to try for extensionless files	—
`+f`	turns a flag on	—
`++f`	increments a flag	—
`-f`	turns a flag off	—
`--f`	decrements a flag	—
`-fallthrough`	ignores switch case fallthrough when used in a lint comment	—
`-father`	a stricter version of `-parent`	—
`-format`	sets the message format for height 3 or less	—
`-format4a`	sets the format of the message that appears above the error for height 4	—
`-format4b`	sets the format of the message that appears below the error for height 4	—
`-format_category`	set format for message category	1.2
`-format_intro`	sets the format of the line that appears before each new message set	—
`-format_stack`	sets the format of the stack usage message	—
`-format_summary`	format of the output produced by the -summary option	—
`-format_verbosity`	sets the format of verbosity output	—
`-function`	copy or remove semantics from *Function0*	—
`+group`	adds messages from *Pattern* to message group *Name*	1.0
`-group`	remove *Pattern* from group *Name* or delete *Name*	1.0
`-h`	adjusts message height options	—
`-header`	auto-include *Filename* at the beginning of each module	—
`--header`	clears previous auto-includes and optionally adds a new one	—
`+headerwarn`	causes message `#829` to be issued when *Filename* is `#included`	—
`-help`	display detailed help about *Option*	1.0
`+html`	emit HTML output	—
`-i`	add search *directory* for `#include` directives	—
`--i`	add lower-priority search *directory* for `#include` directives	—
`-ident`	add identifier characters	—
`-idlen`	specifies the number of meaningful characters in identifier names	—
`-incvar`	change the name of the `INCLUDE` environment variable to *Name*	—
`-index`	establish *ixtype* as index type	—
`-indirect`	process *File* as an options file	—
`-lang_limit`	specify minimum language translation limits	1.0
`+libclass`	add class of headers treated as libraries	—
`+libdir`	specify a *Directory* of headers to treat as libraries	—
`-libdir`	specify a *Directory* of headers to not treat as libraries	—
`+libh`	specify *Headers* to treat as libraries	—
`-libh`	specify *Headers* to not treat as libraries	—
`+libm`	specify *Modules* to treat as libraries	—
`-libm`	specify *Modules* to not treat as libraries	—
`-library`	indicates the next source module is to be treated as library code	—
`++limit`	locks in the message limit at n	—
`-limit`	set the message limit to n	—

Option	Summary	Version
`+lnt`	add indirect file extension(s)	—
`-lnt`	remove indirect file extension(s)	—
`-max_threads`	set the maximum number of concurrent threads for parallel analysis	1.0
`+message`	emits a custom message with the specified message `#`	—
`-message`	emits a custom message via info `865`	—
`+misra_interpret`	enable MISRA interpertation	1.2
`-misra_interpret`	disable MISRA interpertation	1.2
`+oe`	redirect stderr to *Filename* in append mode	—
`-oe`	redirect stderr to *Filename* overwriting existing content	—
`+os`	redirect stdout to *Filename* in append mode	—
`-os`	redirect stdout to *Filename* overwriting existing content	—
`-p`	just preprocess	—
`+paraminfo`	include parameter information for specified messages as verbosity	1.0
`-paraminfo`	exclude parameter information for specified messages as verbosity	1.0
`-parent`	augment strong type hierarchy	—
`+pch`	designates a given header as the pre-compiled header, forcing recreation	—
`-pch`	designates a given header as the pre-compiled header, creating precompiled form if needed	—
`-pp_sizeof`	set the value that `sizeof(`*Text*`)` evaluates to in a preprocessor directive	1.0
`+ppw`	enable preprocessor keyword(s)	—
`-ppw`	disable preprocessor keyword(s)	—
`--ppw`	remove built in meaning of preprocessor keyword(s)	—
`-ppw_asgn`	assign preprocessor word meaning of *Word2* to *Word1*	—
`+pragma`	associates *Action* with *Identifier* for `#pragma`	—
`-pragma`	disables pragma *Identifier*	—
`-printf`	specified *names* are `printf`-like functions with format provided in the *N*th argument	—
`-restore`	restores the state of error inhibition settings	—
`+rw`	enable reserved word(s)	—
`-rw`	disable reserved word(s)	—
`--rw`	remove built in meaning of reserved word(s)	—
`-rw_asgn`	assigns reserved word meaning of *Word2* to *Word1*	—
`-s`	set the size of various types	—
`-save`	saves the current state of error inhibition settings	—
`-scanf`	specified *names* are `scanf`-like functions with format provided in the *N*th argument	—
`-sem`	associates the semantic *Sem* with *Function*	—
`-setenv`	set environment variable *name* to *value*	—
`-size`	set static or auto size thresholds	—
`-skip_function`	skips the body of a *Function* when parsing	1.0
`-specific_climit`	maximum number of specific calls per function	—
`+stack`	enable stack reporting	—
`-stack`	set stack reporting options	—
`-std`	specifies the C or C++ language version	1.0
`-strong`	imbues typedefs with strong type checking characteristics	—
`-subfile`	process just options or just modules from options file *File*	—
`-summary`	outputs a message summary at the end of processing, optionally to a file	—

Option	Summary	Version
-t	sets tab width	—
-tr_limit	set the template recursion limit to n	—
+typename	includes the types of symbols when emitting specified messages	—
-typename	excludes the types of symbols when emitting specified messages	—
-u	undefine the preprocessor symbol *Name*	—
--u	ignore past and future `#define`s of the preprocessor symbol *Name*	—
-unit_check	unit checkout	1.0
--unit_check	unit checkout and ignore modules in lower .lnt files	1.0
-unreachable	ignores unreachable code when used in a lint comment	—
+v	output verbosity to stderr and stdout	—
-v	turn off verbosity or send it to stdout	—
-verbosify	print *string* as a verbosity message	1.0
-vt_depth	specifies the maximum number of nested specific walks	1.0
-vt_passes	specifies the number of passes for intermodule value tracking	1.0
-w	sets the base warning level	—
-width	sets the maximum output width and indentation level for continuations	—
-wlib	sets the base warning level for library code	—
-write_file	write *String* to file *Filename*	1.0
+xml	activates XML escape sequences	—
-zero	sets exit code to 0	—
+zero_err	specify message numbers that should not increment exit code	—
-zero_err	specify message numbers that should increment exit code	—

4.3 Message Options

4.3.1 Error Inhibition

`-e#` disables a message where `#` is a message number or pattern
`+e#` re-enables message(s) `#`

For example, `-e504` will turn off error message `504`. The number designator may contain the wild card characters `?` (single character match) or `*` (multiple character match). For example `-e7??` will turn off all `700` level errors.[2]

As another example:

```
-e1*
```

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

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

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

[2] To turn off Informational messages it is better to use `-w2`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

then,

```
Suppress(715)
```

will suppress message 715 for the next statement or declaration.

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

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

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

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

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

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

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

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

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

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

For example,

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

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

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

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

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

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

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

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

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

```
-e
!e
-ecall
-efile
-efunc
-egrep
-elib
-elibcall
-elibmacro
```

```
-elibsym
-emacro
-estring
-esym
-etype
-w
-wlib
```

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

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

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

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

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

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

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

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

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

The effects of `+efreeze`, `-efreeze`, and `++efreeze` can also be applied to individual message numbers or warning levels by supplying these as arguments to the parameterized version of the options. For example, `++freeze(534)` will prevent later-appearing options from suppressing message `534` without affecting the frozen status of other messages. To inhibit suppression of error messages via new suppression options, `++efreeze(w1)` can be used.

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

For example:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#endif comment
```

instead of

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

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

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

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

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

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

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

Please note that:

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

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

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

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

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

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

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

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

```
-elibsym( 1790 )
```

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

Please note that:

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

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

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

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

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

will suppress message 778 when expanding macro `TROUBLE`.

Please note that

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

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

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

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

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

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

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

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

  would be as if we had defined `ZERO` as:

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

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

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

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

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

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

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

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

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

to be prepended to the macro.

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

Consider the following example:

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

This will result in the following message:

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

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

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

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

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

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

This is one of the more useful options because it inhibits messages with laser-like precision. For example `-esym(714,alpha,beta)` will inhibit error message **714** for symbols `alpha` and `beta`. Messages that are parameterized by the identifier *Symbol* can be so suppressed. Also, if the `fsn` flag is ON (See Section 4.10 Flag Options) messages parameterized by *String* may also be suppressed. Thus, if you examine Message **714** you will notice that the italicized word '*Symbol*' appears in the text of the message.

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

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

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

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

the resulting messages would include:

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

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

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

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

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

to suppress errors about `X::f`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

will remove 2400 from the `formats` group.

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

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

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

```
-group(formats)
```

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

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

`-restore` restores the state of error inhibition settings

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

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

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

Like `-save` there are two forms of the `-restore` option. An inner `-restore` is placed in a `/*` comment. An outer `-restore` is placed outside any module. An inner `-restore` will restore from the last inner `-save`. An outer `-restore` will restore from the last outer `-save`.

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

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

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

A `-save` option can be given within a module (with a `/*lint` comment) or outside a module. We call the first an inner `-save` and the latter an outer `-save`.

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

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

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

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

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

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

The outer `save/restore` facility saves and restores exactly the same error suppression parameters as the inner `save/restore` facility.

`-save/-restore` options that appear in source files are unrelated to `-save/-restore` options that appear outside of source files. Inside a module, it is impossible to `-restore` back to an error state that was saved outside the module.

An implicit outer `-save` occurs at the beginning of all processing; also, an implicit inner `-save` occurs at the beginning of processing for each module.

If you have more `-restore` options than `-save` options within a module, then the extra `-restore` options will revert the error state back to what was in effect at the beginning of processing for that module.

Similarly, if you have more `-restore` options than `-save` options outside of a module, then the extra `-restore` options will revert the error state back to what was in effect at the beginning of all processing.

`-w#` sets the base warning level

The warning levels are:

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

The default warning level is level 3.

The option `-w#` will establish a new warning level and affect only those messages in the "zone of transition". 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(#) sets the base warning level for library code

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

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

For example,

```
-wlib(2)
```

is equivalent to

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

but easier to type.

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

```
-wlib(1)
```

to inhibit all library messages except syntactic errors.

4.3.2 Verbosity

-v[acehiotuw#][mf<*int*>] turn off verbosity or send it to stdout
+v[acehiotuw#][mf<*int*>] output verbosity to stderr and stdout

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

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

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

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

`-v`*n* (where *n* is some integer) will print a message every *n* lines. This option implies `-vf`. This option will also trigger a file resumption message. Example: `-v1` will issue a message for each line of source code.

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

`-vc`... This will cause a message to be printed each time a function is called with a unique set of arguments. This is referred to as a Specific Call. See Section 8.8 Interfunction Value Tracking

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

`-vf` Print the names of all source Files as they are encountered. This means all headers as well as module names. Thus, `-vf` implies `-vm`. This option will indicate which headers are "Library Header Files". See Section 5.1 Library Header Files.

`-vh`... At termination of processing the strong type Hierarchy be dumped in an elegant tree diagram. See the example in Section 7.5.1 The Need for a Type Hierarchy.

`-vh-`... The '`h`' verbosity flag continues to mean that the strong type hierarchy is dumped upon termination. If the '`h`' is followed immediately by a '`-`' then the hierarchy will be compressed, producing the same tree in half the lines. See Section 7.6 Printing the Hierarchy Tree for an example.

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

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

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

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

`-vu`... This verbosity option causes user-defined verbosity messages to be emitted when using the `-verbosify` option.

`-vw`... This verbosity flag will issue a report whenever a function is to be processed with specific arguments. This is called a Specific Walk. See Section 8.8 Interfunction Value Tracking.

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

For example:

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

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

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

`-verbosify(`*string*`)` print *string* as a verbosity message

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

4.3.3 Message Presentation

`-append(#,` *String*`)` append *String* to diagnostic `#` when issued

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

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

will append the indicated message to the text of message 936.

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

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

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

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

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

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

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

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

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

`-format` sets the message format for height 3 or less
`-format4a` sets the format of the message that appears above the error for height 4
`-format4b` sets the format of the message that appears below the error for height 4

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

`%f` = the filename (the `+ffn` flag controls whether full path names are used)
`%l` = the line number
`%t` = the message type (Error, Warning, etc.)
`%n` = the message number
`%m` = the message text
`%c` = the column number (bytes from beginning of line)

`%C` = the column number +1
`%%` = a percent sign
`%(...%)` = conditionally include the information denoted by `...` if location information is available
`\n` = newline
`\t` = tab
`\s` = space
`\a` = alarm (becomes ASCII 7)
`\q` = quote ("")
`\\` = backslash ('\')
`\T` = introduce a real tab into the output
`\e` = ASCII escape

For example the default message format is

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

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

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

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

Sample configuration for colors in terminals supporting ANSI escape codes:

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

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

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

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

For example, pclp -message=hello will emit:

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

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

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

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

A message set consists of a primary message and any supplemental messages that are given in association with the message. -format_intro can be used to produce a line that appears before every new message set, an empty value (the default) results in no introduction line being printed. For example, to separate every message set with 4 dashes, you can use -format_intro=----\n. The same escape sequences supported by -format can be used with -format_intro.

-format_stack sets the format of the stack usage message

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

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

The format string may contain the following escapes:

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

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

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

```
-"format\_stack=%-20f %5a %-20t %5n %c %e"
```

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

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

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

The available format specifiers are:
`%n` = the message Number
`%c` = the Count of instances of a message
`%t` = the message Type
`%m` = the Message text

The default summary format is:

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

`-format_verbosity` sets the format of verbosity output

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

The format string may contain the following escapes:
`%m` = the normal verbosity message

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

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

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

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

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

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

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

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

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

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

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

Example (`-hb^4`):

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

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

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

Message Example (`-hb^3`):

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

Message Example (`-hB^3`):

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

Message Example (`-ha_3`):

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

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

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

Example (`-h$2`):

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

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

Example (`-h1`):

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

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

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

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

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

The sub-options are:

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

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

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

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

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

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

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

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

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

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

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

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

#define MAC 100
MSG(MAC)
```

will result in message `865` being emitted with the text:

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

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

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

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

For each message number equal to or matching #, this option will cause PC-lint Plus to print verbosity information about each parameter cited in the specified message. Example:

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

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

will elicit the message:

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

Message 9272 contains several parameters of different types. Using `+paraminfo(9272)` provides the following parameterization information immediately before the message is emitted:

```
Parameter info for next message (9272)
    String parameter 1: '1'
    Symbol parameter 1: 'B::foo' of type 'int (int)'
    Symbol parameter 2: 'A::foo' of type 'int (int)'
    Symbol parameter 3: 'meters' of type 'int'
    Symbol parameter 4: 'feet' of type 'int'
```

This can be useful to better understand how messages are parameterized and how suppressions can be applied to specific instances of a message.

The three types of message parameters reported are *String*, *Symbol*, and *Type*. For *Symbol* parameters with a type, this type is also reported. The parameters reported by `+paraminfo` are suitable for use in `-estring`, `-esym`, and `-etype` options. For example, any of the below options will work to suppress this instance of message 9272:

```
-estring(9272, 1)              // the referenced parameter number
-esym(9272, B::foo)            // the first symbol referenced
-esym(9272, A::foo)            // the second symbol referenced
-esym(9272, meters)            // the third symbol referenced
-esym(9272, feet)              // the fourth symbol referenced
-etype(9272, int (int))        // the type of the first two symbols
-etype(9272, int)              // the type of the last two symbols
```

-summary | -summary=*filename* outputs a message summary at the end of processing, optionally to a file

This option causes a summary of all issued messages to be presented after Global Wrap-up processing. If a filename is specified, the output is sent to the named file. If not, output is sent to the same output stream used for normal Lint messages (that is, the one specified by -os or, if no such option was issued, stdout).

For each message issued, the summary information consists of the message number (e.g. 1736 for "redundant access specifier"), the number of times that the message was issued, the message type (e.g., "Error", "Warning", etc.) and the message text. This forms a list of all Lint messages that were issued. The list is preceded by a row of column labels to aid readability.

See -format_summary.

-t# sets tab width

Sets the tab width used for indentation checking. The default tab width is 8, but it can be changed to 4 using -t4.

+typename(# [,#...]) includes the types of symbols when emitting specified messages
-typename(# [,#...]) excludes the types of symbols when emitting specified messages

For each message number equal to or matching #, this option will cause PC-lint Plus to add type information for any and all symbol parameters cited in the specified message. Example:

```
class A{};
void g(A a) {}
// Lint reports "Info 1746: parameter 'a' in function 'g(A)'
// could be made const reference"
//lint +typename(174?)
void f(A a) {}
// Lint reports "Info 1746: parameter 'a' of type 'A' in function 'f(A)'
// of type 'void (A)' could be made const reference"
```

One of the purposes of this option is to show the user an exact type name to use as an argument to -etype(). See also +paraminfo.

-width(# [,*Indent*]) sets the maximum output width and indentation level for continuations

An example of the width option is:

```
-width(99,4)
```

The first parameter specifies the width of messages. Lines greater than this width are broken at blanks. A width of 0 implies no breaking. The second number specifies the indentation to be used on continued lines. -width(79,4) is the default setting.

+xml([*tag*]) activates XML escape sequences

By adroit use of the -format option you may format output messages in xml. See the file env-xml.lnt for an example. This option has two purposes. Special xml characters ("<", ">" "&" " " ", and at this writing) will be escaped (to "<", ">" and "&", """, and '" respectively) when they appear in the variable portion of the format. Secondly, if *tag* is not null, the entire output will be bracketed with <*tag*> ... </*tag*>. If *tag* is null this bracketing will not appear.

It is also possible to add a tag in angle brackets that could be used to define, for example, the version of `xml`. Thus:

```
+xml(?xml version="1.0" ?)
+xml(doc)
```

will produce as a prefix the following two lines.

```
<?xml version="1.0" ?>
<doc>
```

Then, at the end of all the message output the following one line will appear.

```
</doc>
```

4.4 Processing Options

4.4.1 Compiler Adaptation

`-A={C|C++ }{`*`YY|YYYY`*`}` specifies the C or C++ language version

Specifies the C or C++ language version. This option is deprecated, please use the `-std` option instead.

```
-A(C90) // specifies C 90
-A(C++2003) // specifies C++ 2003
```

The only languages permitted to be specified are C and C++. This is followed either by a two digit year or a four digit year. It is not necessary to specify the precise year of a standard. For C, any year that precedes 1999 is assumed to be specifying the C90 standard. Years between 1999 and 2010 specify C99, etc. For C++ any year preceding 2011 is assumed to be specifying the 2003 standard, 2011-2013 specifies C++11, 2014-2016 specifies C++14, 2017 and after specifies C++C17. By default we will always assume the latest standard. You may, of course, specify the versions of both C and C++. The language is deduced from the file name extension and not from this option.

`-lang_limit(C|C++ , `*`limit-name`*`, `*`limit-value`*`)` specify minimum language translation limits

The `-lang_limit` option allows customization of the minimum translation limits reported by message `793`. This option can be used to increase, decrease, or restore the default limits for individual categories when processing C and C++ code.

The first argument is C or C++ indicating which language the change should affect. The second argument is the name of the limit to modify. The third argument is either a non-negative integral value or the special value `default` indicating that any customized value should be removed, restoring the default for the language. For example:

```
-lang_limit(C++, function_arguments, 10)
```

Will cause message `793` to be emitted whenever a function call with more than 10 arguments is encountered in a C++ file. A value of `0` for *`limit-value`* can be used to indicate the lack of a limit. For a list of the supported limits, default values, and limit names for use with this option, see the 11.10 Language Limits section.

`-std={c89|c90|c99|c11|c++03|c++11|c++14|c++17}` specifies the C or C++ language version

The supported values for this option are: `c89`, `c90`, `c99`, `c11`, `c++03`, `c++11`, `c++14`, and `c++17`. Unlike the `-A` option, this option requires one of the above values, e.g. neither `c++2011` nor `C++13` is a valid way to specify `C++11` support.

4.4.2 Preprocessor

`-d{Name}[={Value}]` define the preprocessor symbol *Name* with value *Value*

This option allows the user to define preprocessor variables (and even function-like macros) from the command line. The simplified format of this option is:

```
-dName[=Value]
```

where the square brackets imply that the value portion is optional. If `=Value` is omitted, 1 is assumed. If only *Value* is omitted as in `-dX=` then the value assigned is null. For alternative syntax allowing easier use of string literals in the replacement, use `-dName{Replacement}`. Examples:

```
-dDOS
-dalpha=0
-dX=
```

These three options are equivalent to the statements

```
#define DOS 1
#define alpha 0
#define X
```

appearing at the beginning of each subsequent module.

Note that case is preserved. There is no intrinsic limit to the number of `-d` options. See also the `-u`... option.

This option does not provide any functionality over what can be provided through the use of `#define` within the code. It does allow lint to be customized for particular compilers without modifying source. It also applies globally across all modules, whereas `#define` is local to a specific module.

`+dName[=Value]` define the preprocessor symbol *Name* resistent to change via `#define`
`++dName[=Value]` define the preprocessor symbol *Name* that cannot be `#undef`'d

For added security `++dName=Value` will behave in a similar fashion and, moreover, name cannot be `undef`'ed.

Using this option you can lock in the definition of function-like macros as well as object macros.

For example, suppose the PC-lint Plus is stumbling badly over the macro

```
offsetof(s,m)
```

First place your definition within a header file under a slightly different name:

```
#define my_offsetof(s,m) some definition...
```

Then use the options:

```
+doffsetof=my_offsetof
-header( my_offsetof.h )
```

where `my_offsetof.h` contains the definition of the `my_offsetof` macro.

You may also explicitly set function-like macros. See `-dName`.

`-header(`*`Filename`*`)` auto-include *Filename* at the beginning of each module
`--header | --header(`*`Filename`*`)` clears previous auto-includes and optionally adds a new one

This is useful for defining macros, `typedefs`, etc. of a global nature used by all modules processed by PC-lint Plus without disturbing any source code. For example,

```
-header( lintdcls.h )
```

will cause the file `lintdcls.h` to be processed before each module.

The header is not reported as being unused in any given module (even though it may be). It is not considered a library header. An extra option may be needed to make this assertion as follows: `+libh(`*`lintdcls.h`*`)`

Multiple `-header` options may be used, and this effect is additive. Files are included in the order in which they are given. However, an option of the form:

```
--header( Filename )
```

will remove **all** prior headers specified by `-header` options before adding *Filename*.

If *Filename* is absent as in `--header` then the effect is to erase **all** prior `-header` requests.

`-i`*`directory`* add search *directory* for `#include` directives
`-I`*`directory`* add search *directory* for `#include` directives

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

```
-i/lib/
```

can be used to make sure that all files not found in the current directory are looked up in some library directory named `lib`. A directory separation character will be appended onto the end of the `-i` option if not already present. Thus

```
-i/lib
```

is equivalent to the above.

To include blanks within the directory name employ the quote convention (See Section 4.1 Rules for Specifying Options) as in the following:

```
-i"program files\{compiler}"
```

Multiple directories may be specified either with multiple `-i` options or by specifying a semi-colon separated list with a single `-i` option. (See also `+fim`)

PC-lint Plus also supports the INCLUDE environment variable, See Section 12.2.1 INCLUDE Environment Variable. Note: Any directory specified by a `-i` directive takes precedence over the directories specified via the INCLUDE environment variable.

`-I`*`directory`* is identical to `-i`*`directory`*.

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

`--i`*`directory`* add lower-priority search *directory* for `#include` directives

All directories specified by `-i` are searched before directories named by `--i`. This is to support compilers that always search through compiler-provided library header directories after searching user-provided directories.

Example: suppose there is a header file named 'bar.h' in both directory '/foo' and directory 'local'. Then:

```
// in std.lnt:
--i/foo        // search foo with low priority
-ilocal        // search local with high priority
// in t.cpp:
#include <bar.h> // finds the version in 'local'
```

Thus the priority of the `--i` option is always lower than the `-i` option. How does it compare with the `INCLUDE` environment variable, which is also lower than `-i`? If the `--i` option is in an indirect file (`.lnt` file) it will act as though it had a lower priority than the `INCLUDE` environment variable. This is because the `INCLUDE` variable is triggered upon reading the first file. If the `--i` option is on the command line or in the `LINT` environment variable it will take priority over the `INCLUDE` variable.

`-incvar(`*`Name`*`)` change the name of the `INCLUDE` environment variable to *Name*

The environment variable `INCLUDE` is normally checked for a list of directories to be searched for header files (See Section 12.2 include Processing). You may use the `-incvar(`*`Name`*`)` option to specify a variable to be used instead of `INCLUDE`. For example

```
-incvar(MYINCLUDE)
```

requests checking the environment variable `MYINCLUDE` rather than checking `INCLUDE`.

Limitation: This option may not be placed in an indirect `.lnt` file or source file. It may be placed on the command line or within the `LINT` environment variable. The `INCLUDE` environment variable is processed just before opening the first file.

`-pch(`*`Header`*`)` designates a given header as the pre-compiled header, creating precompiled form if needed
`+pch(`*`Header`*`)` designates a given header as the pre-compiled header, forcing recreation

The *header name* should be that name used between angle brackets or between quotes on the `#include` line. In particular, if the name on the `#include` line is not a full path name do not use a full path name in the option. See Section 6.2 Designating the precompiler header.

`-pp_sizeof(`*`Text, Value`*`)` set the value that `sizeof(`*`Text`*`)` evaluates to in a preprocessor directive

This option is provided for legacy code and will direct PC-lint Plus on how to evaluate a particular `sizeof` expression appearing in a preprocessor conditional. See 12.6 Preprocessor sizeof.

`-u`*`Name`* undefine the preprocessor symbol *Name*

For example:

```
-u_lint
```

will undefine the identifier `_lint`, which is normally pre-defined before each module. The undefine will take place for all subsequent modules after the default pre-definitions are established. The observant reader will notice that you may not undefine the name `nreachable`.

`--u`*`Name`* ignore past and future `#define`s of the preprocessor symbol *Name*

```
//lint --uX
#define X 1
int y = X;
```

will be equivalent to:

```
int y = X;
```

Please note the difference between this option and the `-u`*Name* option, which undefines any built-in definition for *Name* but does not affect definitions that *Name* may acquire in the future.

`+ppw(`*word* `[,`*word*...`])` enable preprocessor keyword(s)
`-ppw(`*word* `[,`*word*...`])` disable preprocessor keyword(s)

This option removes any predefined meaning we may associate with the preprocessor word(s) (*Word1*, *Word2* etc.). If this is followed by a `+ppw(`*word*`)` the word is entered as a no-op rather than one that has a predefined meaning. For example, if your code contains the non-standard preprocessor directive `#include_next` and if its meaning coincides with that of the GNU compiler, then just issue the option `+ppw(include_next)`. However, if you would rather have it ignore such a preprocessor word, issue the commands:

```
--ppw(include_next)
+ppw(include_next)
```

`--ppw(`*word* `[,`*word*...`])` remove built in meaning of preprocessor keyword(s)

If this is followed by a `+ppw(`*word*`)` the word is entered as a no-op rather than one that has a predefined meaning. For example, if your code contains the non-standard preprocessor directive `#dictionary` and if its meaning coincides with that of the DEC VMS compiler, then just issue the option `+ppw(dictionary)`. However, if you would rather have it ignore such a preprocessor word, issue the commands:

```
--ppw(dictionary)
+ppw(dictionary)
```

`-ppw_asgn(`*Word1*`,` *Word2*`)` assign preprocessor word meaning of *Word2* to *Word1*

This option assigns the preprocessor semantics associated with *Word2* to *Word1* and activates *Word1*. E.g.

```
-ppw_asgn( header, include )
```

will then make

```
#header <stdio.h>
```

behave exactly like:

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

The purpose of this option is to support special non-standard preprocessor conventions provided by some given compiler.

Even though *Word2* may not be activated it may still have semantics. Thus

```
-ppw_asgn( INC_NEXT, include_next )
```

will assign the semantics associated with the `include_next` preprocessor directive to `INC_NEXT` and activate `INC_NEXT`; all this in spite of the fact that `include_next` has not been activated (with the `+ppw` option). See Section 12.4 Non-Standard Preprocessing for descriptions of non-standard preprocessing directives.

The `#macro` pre-processing directive is not a directive implemented by any compiler to our knowledge. So why, you ask, are we providing this directive? We are providing `#macro` as a way for programmers to implement arbitrary preprocessor directives by converting directives into macros.

For example, one compiler accepts the following preprocessor directive

```
#BYTE n = 'a'
```

as a declaration of the variable `n` having a type of `BYTE` and an initial value of 'a'. The `#macro` directive will allow us to express `#BYTE` as a macro and so render the directive as C/C++ code.

The preprocessing directive:

```
#macro a b c
```

will result in the macro invocation

```
sharp_macro( a b c )
```

The word `sharp` is used as a prefix because the word 'sharp' is often used to denote verbally the '#' character.

We can transfer the properties of `#macro` to some other actual or potential preprocessing directive using the option `ppw_asgn`. For example:

```
-ppw\_asgn( BYTE, macro )
```

will assign the `#macro` properties to `BYTE` (and also enable `BYTE` as a preprocessing directive). Then the directive

```
#BYTE n = 'a'
```

will result in the macro call:

```
sharp_BYTE( n = 'a' )
```

Presumably there is a macro definition that resembles:

```
#define sharp_BYTE(s) unsigned char s;
```

Such a definition can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

`+pragma(`*`Identifier, Action`*`)` associates *Action* with *Identifier* for `#pragma`
`-pragma(`*`Identifier`*`)` disables pragma *Identifier*

The `+pragma(`*`identifier, Action`*`)` option can be used to specify an *identifier* that will be used to trigger an *action* when the *identifier* appears as the first identifier of a `#pragma` statement. See Section 4.11.7 User pragmas.

4.4.3 Tokenizing

`-$` permit `$` in identifiers

`-ident(`*`chars`*`)` add identifier characters

This option allows the user to specify alternate identifier characters. Each character in *chars* is taken to be an identifier character. For example if your compiler allows `@` as an identifier character then you may want to use the option:

```
-ident(@)
```

Option `-$` is identical in effect to `-ident($)` and is retained for historical reasons.

`+rw(`*word* `[,`*word*`...])` enable reserved word(s)
`-rw(`*word* `[,`*word*`...])` disable reserved word(s)

If the meaning of a reserved word being added is already known, that meaning is assumed. For example, `+rw(typeof)` will enable the reserved word `typeof`. If the reserved word has no prior known semantics, then it will be passed over when encountered in the source text. As another example:

```
+rw( __inline, entry )
```

adds the two reserved words shown. `__inline` is assigned a meaning consistent with that of the Microsoft C/C++ compiler. `entry` is assigned no meaning; it is simply skipped over when encountered in a source statement. Since no meaning is to be ascribed to `entry`, it could just as well have been assigned a null value as in

```
-dentry=
```

`--rw(`*word* `[,`*word*`...])` remove built in meaning of reserved word(s)

If *word* has known semantics, remove those semantics. For example, `+rw(global)` installs the reserved word `global` with the meaning it has in OpenCL. If you don't want that meaning but would rather have `global` ignored, then use the option sequence:

```
--rw(global)
+rw(global)
```

`-rw_asgn(`*Word1*`,` *Word2*`)` assigns reserved word meaning of *Word2* to *Word1*

assigns the keyword semantics associated with *word2* to *word1* and activates *word1*. E.g.

```
-rw_asgn( interrupt, _to_brackets )
```

will assign the semantics of `_to_brackets` to `interrupt`. This will have the effect of ignoring `interrupt(21)` in the following:

```
void f( int n ) interrupt(21) { }
```

The purpose of this option is to support special non-standard keyword conventions provided by some given compiler. But do not overlook the use of the `-d` option in this connection. `-d` (or the equivalent `#define`) can be more flexible since a number of tokens may be associated with a given identifier.

4.4.4 Parsing

`-fallthrough` ignores switch case fallthrough when used in a lint comment

indicates that the programmer is aware of the fact that flow of control is falling through from one case (or default) of a switch to another. Without such an option Message 825 will be issued. For example:

```
case 1:
case 2: n=0;      //setting n to 0
case 3: n++;
```

will result in Info 825 on case 3 because control is falling through from the statement above, which is neither a case nor a default. The cure is to use the `-fallthrough` option:

```
case 1:
case 2: n = 0;    //setting n to 0
//lint -fallthrough
case 3: n++;
```

Warning 616 will be issued if no comment at all appears between the two cases. If this is adequate protection, then just inhibit message 825.

`-unreachable` ignores unreachable code when used in a lint comment

This is useful to inhibit some error messages. For example, suppose `my_exit()` does not return. Then:

```
int f(n)
    {
    if(n) return n;
    my_exit(1);
    //lint -unreachable
    }
```

contains an unreachable indicator to prevent PC-lint Plus from thinking that an implied return exists at the end of the function. An implied return would not return a value but `f()` is declared as returning `int`. Note, however, that it would have been better practice to copy the exit semantics to `my_exit`. Eg.:

```
-function(exit,my_exit)
```

In this case the `-unreachable` option would not have been necessary.

4.4.5 Template

`-tr_limit=`*n* set the template recursion limit to *n*

This option allows the user to specify a Template Recursion limit. When the limit is reached, message 1777 is issued, which reminds you that you may use this option to deepen the level of recursion. See message 1777 for further details.

4.5 Data Options

4.5.1 Scalar Data Size

`-s` set the size of various types

This option allows setting the size of various scalars (`short`, `float`, pointers, etc.) for the target machine. The default sizes are for a 32-bit target machine following the ILP32 model. If you are targeting a 32-bit system that uses a different data model, targeting a 64-bit architecture, or targeting an embeddded environment, then you will need to adjust type sizes to match.

For example, an LP64 target would use the size options:

```
-ss2 -si4 -sl8 -sll8 -sp8
```

and an embedded system with 16-bit characters might use:

```
-sb16 -ss1 -si1 -sl1 -sll2 -sp2
```

Note that the maximum size that can be specified for standard types is 64 bits. Because the size of `long long` defaults to 8 bytes, any increase in the size of a byte will require a decrease in the size of `long long` or a size option misconfiguration error will be issued for `long long`.

In the list below `#` denotes a small positive integer that represents the size in characters for the corresponding type (except for `-sb#` where `#` is in units of bits).

`-sb#` The number of bits in a `char` is `#`. The default is 8. Note that by the definition of the `sizeof` operator, `sizeof(char) == 1` regardless of `#`. This value is conceptually similar to the standard macro `CHAR_BIT`, although your standard library must still define this correctly in a way that matches the value set using this option.

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

`-ss#` `sizeof(short)` becomes `#`. The default is 2.

`-si#` `sizeof(int)` becomes `#`. The default is 4.

`-sl#` `sizeof(long)` becomes `#`. The default is 4.

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

`-sf#` `sizeof(float)` becomes `#`. The default is 4.

`-sd#` `sizeof(double)` becomes `#`. The default is 8.

`-sld#` `sizeof(long double)` becomes `#`. The default is 8.

`-sp#` size of pointers become `#`. The default is 4. There must be at least one integer type size that matches the pointer size.

`-sw#` `sizeof(wchar_t)` becomes `#`. The default is 2. Note this only affects a built-in `wchar_t`.

See `-a` to specify alignment of types and the relationship between size and alignment.

`-size(`*`flags`*`, `*`amount`*`)` set static or auto size thresholds

This option causes an Informational message (812, 813, 2712 or 2713) to be issued whenever a data variable's size in bytes equals or exceeds a given *amount*. *Flags* can be used to indicate the kind of data as follows:

- `s` static data (Info 812). Data can be file scope (`extern` or `static`) or declared `static` and local to a function.
- `a` auto data (i.e., stack data) (Info 813). See also `-stack`, which can do a comprehensive stack analysis.
- `p` parameter variable (Info 2712). Checks function parameters for types of a size that equal or exceed the given amount.
- `r` return value (Info 2713). Checks for functions that return a type with a size that equal or exceed the given amount.

The purpose of the `-size` option is to detect potential causes of stack overflow (using the '`a`' flag); to flag large contributors to excessively large static data areas (using the '`s`' option); to identify functions that are receiving (using the '`p`' option) or returning (using the '`r`' option) large data types that may be more appropriate to pass by pointer or reference.

E.g. `-size(a,100)` detects auto variables that equal or exceed 100 bytes. If you have a stack overflow problem, such a test will let you focus on a handful of functions that may be causing the overflow. It does not, however, look at call chains and does not compute an overall stack requirement either of a single function or of a sequence of calls.

If *amount* is 0 (it is by default) no message is given.

4.5.2 Scalar Data Alignment

`-a` set the alignment of various types

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

Alignment has only minimal impact on the behavior of PC-lint Plus. At this writing there are only three messages (958, 959 and 2445) that detect alignment irregularities. But in addition to these it is sometimes essential to get alignment correct in order to know the sizes of compound data structures.

For every size option (see `-s`) of the form:

`-s`*`Type#`*

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

`-a`*`Type#`*

For example, the option

```
-ai1
```

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

If an alignment for a type is not explicitly given by a `-a` option, an alignment is deduced from the size of the type. The deduced alignment is the largest power of 2 that evenly divides the type. Thus if the size is 4 the deduced alignment is 4 but if the size is 6 the deduced alignment is 2. This deduction is made just before the first time that alignment (and size) may be needed. An attempt to use the `-a`*`Type`* option (or the `-s`*`Type`* option) after this point is greeted with Error 686. For example:

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

4.6 Miscellaneous Options

4.6.1 File

`+cpp(`*`Extension`*` [,`*`Extension`*`...])` add C++ extension(s)
`-cpp(`*`Extension`*` [,`*`Extension`*`...])` remove C++ extension(s)

This option allows the user to add and/or remove extensions from the list that identifies C++ modules. By default only `.cpp` and `.cxx` are recognized as C++ extensions. (It is as if `+cpp(cpp,cxx)` had been issued at the start of processing.) For example:

```
lint a.cpp +cpp(cc) b.cc c.c
```

treats `a.cpp` and `b.cc` as C++ modules and `c.c` as a C module. There is no intrinsic limit to the number of different extensions that can be used to designate C++ modules. See also flag `+fcp`.

Note: If you are using `+cpp(.C)`, i.e. you want to use case to distinguish C++ vs. C on Windows, you need to also turn off the fold file name flag (`-fff`).

`+ext(`*`Extension`* `[,`*`Extension`*`...])` set the extensions to try for extensionless files

For example,

```
lint alpha
```

will, by default, cause first an attempt to open `alpha.lnt`. If this fails there will be an attempt to open `alpha.cpp`. If this fails there will be an attempt to open `alpha.cxx`. Finally, an attempt to open `alpha.c` will be made. It is as if the option:

```
+ext( lnt, cpp, cxx, c )
```

had been given on startup.

Minor notes: This has no effect on which extensions indicate that a module is to be regarded as a C++ module. This is done by the options `-/+cpp` and `-/+fcp`. Prefixing an extension with a period has no effect. Thus, `+ext(lnt,c)` means the same as `+ext(.lnt,.c)` . For Windows, upper-casing the extension also has no effect. Thus, `+ext(lnt)` has the same effect as `+ext(LNT)`. On Unix, however, case differences do matter. For example, if the Unix programmer wanted both `.c` and `.C` extensions to be taken by default he might want to use the option: `+ext(lnt,c,C)`.

`+headerwarn(`*`Filename`*`)` causes message `#829` to be issued when *Filename* is `#included`

For example `+headerwarn(stdio.h)` will alert the programmer to the use of `stdio.h`. If the option `-wlib(1)` is in place, as it usually is to stem the flood of Warnings and Informationals emanating from library headers, no message `829` will be issued from within a library header unless you also issue a `+elib(829)` sometime after the `-wlib(1)`.

`-indirect(`*`File`* `[,...])` process *File* as an options file

Allows you to specify Lint option files to be processed when this option is encountered. This is useful if you want to use an options file within a Lint comment. For example, Lint option files that are appropriate only for a particular configuration may be conditionally included. The code below may appear in some header that is included either in every module or at least the first module. Thus, the header in question can be injected with the `-header` option.

```
#ifndef BEEN_HERE
#define BEEN_HERE
   #if defined(HAS_LIBRARY_A)
       //lint -indirect(lib-a.lnt)
   #elif defined(HAS_LIBRARY_B)
       //lint -indirect(lib-b.lnt)
   #else
       //lint -indirect(lib-default.lnt)
   #endif
#endif
```

`+libclass(`*`Identifier`* `[,...])` add class of headers treated as libraries

This option specifies the class of header files that are by default treated as library headers. Arguments can be one of:

`angle` (specified with angle brackets),
`foreign` (comes from a foreign directory using `-i` or the `INCLUDE` environment variable),
`ansi` (one of those specified by ANSI/ISO C), or
`all` (meaning all header files).

For more information, see Section 5.1 Library Header Files.

`+libdir(`*`Directory`*` [,...])` specify a *Directory* of headers to treat as libraries
`-libdir(`*`Directory`*` [,...])` specify a *Directory* of headers to not treat as libraries

This option allows you to override `+libclass` for specified directories. *Directory* may contain wild cards ('`*`' and '`?`'). For more information, see Section 5.1 Library Header Files.

`+libh(`*`Header`*` [,...])` specify *Header*s to treat as libraries
`-libh(`*`Header`*` [,...])` specify *Header*s to not treat as libraries

This option allows you to override `+libclass` and `+/-libdir` for specified headers. *Header* may contain wild card characters. For more information, see Section 5.1 Library Header Files.

`+libm(`*`Module`*` [,...])` specify *Module*s to treat as libraries
`-libm(`*`Module`*` [,...])` specify *Module*s to not treat as libraries

The *Module* may contain wild card characters. For more information, see 5.1 Library Header Files.

`-library` indicates the next source module is to be treated as library code

This option turns ON the library flag for the next module, if given on the command line, or for the rest of the module if placed within a lint comment. For an example, see Section 5.2 Library Modules. At one time this option was equivalent to the `+flb`. However, there is now a difference. `-library` designates the file in which it is placed and all files that it may include as having the library property. Thus, if a lint option within a header file contains the `-library` flag then only that header (and the headers it includes) are affected. It does not affect the including file. With `+flb` the flag is left on until turned off.

`+lnt(`*`Extension`*` [,`*`Extension`*`...])` add indirect file extension(s)
`-lnt(`*`Extension`*` [,`*`Extension`*`...])` remove indirect file extension(s)

Modidy 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)
```

4.6.2 Global

`?` displays help

`-b` suppress banner output
`+b` redirect banner output to stdout
`++b` produce banner line

(Unlike most other options, this option must be placed on the command line and not in an indirect file.) When PC-lint Plus is run from some environments, the banner line (identifying the version of PC-lint Plus and bearing a Copyright Notice) may overwrite a portion of an editing screen. This is because the banner line is, by default, written to standard error whereas the messages are written to standard out and can be redirected. The option `+b` will cause the banner line to be written to standard out (and hence will become part of the redirected output). The option `-b` will suppress the banner line completely.

The option `+b` works well for:

```
lint +b ... >outfile
```

Unfortunately this will not have the intended effect with:

```
lint +b -os(outfile) ...
```

as the banner line is written before the `-os` option has had a chance to take effect.

`++b` will deposit the banner line into standard out anywhere it is encountered. Thus:

```
lint -os(outfile) ++b ...
```

will cause the banner line to be placed into `outfile`. You will also get a banner line in standard error but this can be separately suppressed as in:

```
lint -b -os(outfile) ++b ...
```

`-cond(`*`conditional-expr, true-options`* `[,`*`false-options`*`])` conditionally execute options

Note: This option exists mainly to support an experimental feature that will be available in a future release.

The `-cond` option accepts two or three arguments, the first of which is a conditional expression to evaluate when the option is processed, the second is the set of options to execute if the conditional expression evaluates to true, and the third (optional) argument specifies the options to execute if the conditional expression is false.

The conditional expression may contain string and numeric comparisons and string pattern matching using regular expressions. Each operand in an expression is either numeric or a string. A string is any text that is surrounded by single quotes, everything else is numeric. Valid numeric values include anything that the standard C function `strtod` can parse as well as the literal values `true` and `false`, which represent the values 1 and 0 respectively.

The arithmetic operators `+`, `-`, `*`, and `/` may be used on numeric values within the expression and possess the same meaning and precedence as their corresponding C operators. The `/` operator performs floating point division, e.g. 1 / 2 will evaluate to 0.5, not 0. The `//` operator performs integer division and has the same precedence as `/`. The `mod` operator performs integer modulus arithmetic and is equivalent to the `%` operator in C. The `fmod` operator yields the floating point modulus value of its operands. The unary operators `+`, `-`, and `!` have the traditional meaning when applied to numeric operands.

The comparison operators `<`, `<=`, `>`, `>=` and the equality operators `==` and `!=` can be used on either numeric or string operands. When used with numeric operands, they behave as the corresponding C operators. When used with string operands, they perform lexicographic comparisons. The operands must be of the same type (numeric or string).

The logical operators `&&` and `||` and the ternary operator `?:` are supported and have the same meaning as the corresponding C operators. Parentheses may be used for grouping subexpressions.

The pattern matching operator `~` takes two operands, a subject string on the left-hand side and a pattern on the right-hand side. Both operands must be strings. The result is true if the subject string matches the regular expression pattern and false otherwise. The PCRE (Perl) regular expression syntax is used for the regular expressions supported by the `-cond` option.

The special identifier `__is_stdout_terminal` evaluates to `1` if standard output does not appear to have been redirected or piped and `0` otherwise.

`-dump_messages(file=`*`filename`* `[,format={plain|list|json|yaml|csv|xml}] [,`*`sub-options`*`])`
dumps PC-lint Plus messages to the provided file in the specified format

This option writes out the PC-lint Plus message list to the filename specified with the `file=`*`filename`* sub-option. The `format` sub-option specifies the format to use when writing messages and may be any of `plain`, `list`, `json`, `yaml`, `csv`, or `xml`. Specifying a format of `json`, `yaml`, `csv`, or `xml` will result in messages being written in the corresponding format (see examples below). A format of `plain` will result in output similar to the `msg.txt` file provided with PC-lint 9. A format of `list` results in a list output containing one message per line with fields separated by tabs. The default format is `plain`.

The `include_commentary` sub-option may be used to indicate whether Reference Manual descriptions of each message should be included in the output. The default is to include commentary; `include_commentary=false` will disable commentary. Note that commentary is not supported for the `list` format and cannot be disabled for the `plain` format.

The `include_clang_errors` sub-option indicates whether messages in the 4*xxx* and 5*xxx* range are included in the output. These messages are mapped clang errors that do not contain descriptions and are excluded from the output by default. Use `include_clang_errors` or `include_clang_errors=true` to include these messages.

In all cases, messages are written in ascending order of message number.

Examples

`-dump_messages(file=msg.txt)` will result in output that looks like:

```
error 1
unclosed comment

    End of file was reached with an open comment still unclosed.


error 2
unclosed quote

    An end of line was reached and a matching quote character (single or
    double) to an earlier quote character on the same line was not found.
```

`-dump_messages(file=msg.txt,format=list)` looks like:

```
1       error   unclosed comment
2       error   unclosed quote
3       error   #elif without a #if
5       error   too many #endif directives
8       error   unclosed #if
9       error   #elif after #else
```

`-dump_messages(file=msg.txt,format=json)` looks like:

```
[
    {
        "ID" : "1",
        "CATEGORY" : "error",
        "TEXT" : "unclosed comment",
        "COMMENTARY" : "End of file was reached with an open comment still unclosed."
    },
    ...
]
```

`-dump_messages(file=msg.txt,format=yaml)` looks like:

```
-
    id: 1
    category: error
    text: 'unclosed comment'
    commentary: |
        End of file was reached with an open comment still unclosed.
-
    id: 2
    category: error
    text: 'unclosed quote'
    commentary: |
        An end of line was reached and a matching quote character (single or
        double) to an earlier quote character on the same line was not found.
```

`-dump_messages(file=msg.txt,format=csv)` looks like:

```
"1","error","unclosed comment","End of file was reached with an open comment still unclosed."
"2","error","unclosed quote","An end of line was reached and a matching quote character (single or
double) to an earlier quote character on the same line was not found."
"3","error","#elif without a #if","A #else was encountered not in the scope of a #if, #ifdef or #ifndef."
```

Note that newlines may be embedded in the commentary field.

`-dump_messages(file=msg.txt,format=xml)` looks like:

```
<messages>
    <message id="1">
        <category>error</category>
        <text>unclosed comment</text>
        <commentary>End of file was reached with an open comment still unclosed.</commentary>
    </message>
    ...
</messages>
```

`-dump_message_list=`*`filename`* dumps PC-lint Plus message list to the provided file

`-dump_message_list` will case PC-lint Plus to write out its list of messages to the provided file. For example, `-dump_message_list(mlist.txt)` will write the message information for all messages supported by PC-lint Plus to a file named `mlist.txt`. This file contains one line per message with three fields, delimited by tabs as shown below:

```
25  error   character constant too long for its type
29  error   duplicated type-specifier, '__detail__'
31  error   redefinition of symbol __symbol__
32  error   field size (member __symbol__) should not be zero
```

Parameterized messages show up in the same way that they do when using the `-summary` option.

`-exitcode=`*n* set the exit code to *n*

By default, PC-lint Plus terminates with an exit code of 0 upon successful completion and an exit code of 1 when terminating prematurely, such as from a fatal error. If the `frz` flag is turned OFF, PC-lint Plus will instead exit with the total number of messages emitted. This option can be used to specify the exit code that PC-lint Plus should return upon completion (successful or otherwise). This option has no effect if the `frz` flag is ON. See also `-zero`, and `+zero_err`.

`+f` turns a flag on
`-f` turns a flag off
`++f` increments a flag
`--f` decrements a flag

See Section 4.10 Flag Options.

`-help=`*Option* display detailed help about *Option*

With no arguments, the `-help` option produces the same output as the `?` option. When provided with the name of an option, help information specific to the provided option is emitted. For example, given `-help=+libh`, the following will be emitted:

```
OPTION:   +libh
GROUP:    Miscellaneous
CATEGORY: File
USAGE:    +libh(Header [,...])

specify Headers to treat as libraries
```

`-max_threads=`*n* set the maximum number of concurrent threads for parallel analysis

The `-max_threads` option can be used to specify the number of concurrent threads (the default is 1, which essentially disables multi-threading). One thread will be created for each source module, up to the specified maximum. The `-max_threads` option must appear before the first module to have any effect. See Section 11.9 Parallel Analysis for more information.

`-p | -p(`*width*`)` just preprocess

If this flag is set, the entire character of PC-lint Plus is changed from a diagnostic tool to a preprocessor. The output is directed to standard out, which may be redirected. Thus,

```
lint -os( file.p ) -p file.c
```

will produce in `file.p` the text of `file.c` after all `#` directives are carried out and removed. This may be used for debugging to determine exactly what transformations are being applied by PC-lint Plus.

The optional argument (*width*) denotes an upper bound on the width of the output lines. For example:

```
-p(100)
```

will limit the width of output lines to 100 characters. Splitting is done only at token boundaries. Very large tokens will not be split even if they exceed the nominal line limit. This is so the result can be passed back through lint or some other C/C++ source processor.

In order to track down some complicated cases involving many include headers you may want to use the `-v1` verbosity option in connection with `-p`. Recall (Section 4.3.2 Verbosity) that `-v1` will produce a line of output for every line processed. When you use both options together, as in, for example:

```
lint -os( file.p ) -v1 -p file.c
```

then the single line will be preceded by the name of the file and the line number both enclosed in a C comment. This will enable you to track through every line of every header processed.

`-setenv(`*`name`*`=`*`value`*`)` set environment variable *name* to *value*

will allow the user to set an environment string. The directive is of the form *name*=*value*. For example:

```
-setenv(ROOT_DIR=\{home}\{program}\{dev})
```

will set the environment variable `ROOT_DIR` to the indicated directory. This can be used subsequently in PC-lint Plus options by using the `%`*`var`*`%` syntax. For example:

```
-iROOT_DIR%\include}
```

establishes a new search directory based on the environment name.

The environment variable setting will last for the duration of the process.

`-skip_function(`*`Function`*` [,`*`Function`*`...])` skips the body of a *Function* when parsing

Causes the bodies of the named functions to be skipped during parsing. Functions whose bodies are skipped cannot be semantically analyzed. This option is useful if you are using compiler-specific syntax that cannot be easily accommodated by PC-lint Plus and such usage is isolated to a handful of functions. In general it is better to configure PC-lint Plus to appropriately handle compiler-specific peculiarities but this option can be used as a last resort. See also the `flf` flag to automatically skip bodies of library functions.

`-subfile(`*`File`*`, options|modules)` process just options or just modules from options file *File*

This is an unusual option and is meant for front-ends trying to achieve some special effect. There are two forms of the option; one with the second argument equal to `options` and the other with the second argument equal to `modules`. In general, indirect files (those ending in `.lnt`) will contain both options and modules. Sometimes it is important to extract just the options from such a file. One example is if you are attempting to do a unit-check on one particular module. Say your project file is `project.lnt`. Then you might do project and unit checks using the same indirect file.

```
lint project.lnt // project check
lint -subfile( project.lnt, options ) filename // unit check
```

Note that `project.lnt` may itself have indirect files and that modules and options may be interspersed. The rule is that every indirect file is followed for as long as it takes until the first module is encountered. Every option thereafter is considered not a general option but specific to project check out.

With `modules` as the second argument to `subfile`, the processing picks up at precisely the point that the '`options`' subargument left off. Thus if you wanted to place a particular option, say `-e1706`, just before the first module of `project.lnt` you could achieve that effect by placing the following in either an indirect file or on the command line:

```
-subfile( project.lnt, options ) -e1706
-subfile( project.lnt, modules )
```

`-unit_check` unit checkout

This is one of the more frequently used options. It is used when linting a subset (frequently just one) of the modules comprising a program and suppresses Global Wrapup and the messages that would be issued during global wrapup analysis. For historical reasons, `-u` is an alias for this option.

`--unit_check` unit checkout and ignore modules in lower .lnt files

This option is like `-unit_check` except that any module at a lower `.lnt` level is ignored. Suppose, for example, that `project.lnt` is a project file containing both options and module names. Then the command line:

```
lint --unit_check project.lnt alpha.cpp
```

will do a unit check on module alpha.cpp. It will ignore any module names that may be identified within `project.lnt`. `project.lnt` does not have to immediately follow the `--unit_check` option. Any `.lnt` file within `project.lnt` will similarly be processed for options but module names will be ignored. See also `-subfile()` which deals with this issue in a more comprehensive manner.

For historical reasons, `--u` is an alias for this option.

`-write_file(`*`String`*`, `*`Filename`*` [,append=true|false] [,binary=true|false])` write *String* to file *Filename*

The `-write_file` option is used to write data to a file. The option accepts a string to write and the name of the file to write the contents of the string to. Backslash escapes appearing in the string are converted as expected. By default, the output file is overwritten, to append data to the end of the file, use the sub-option `append=true`. The sub-option binary can be used to write the data in binary mode. For example, to append a line containing the text "Starting Lint version X" where X is the major version number to a file named "`/home/pclint/lint.log`", the below is used:

```
-write_file("Starting Lint version %LINT_VERSION%\n",
     "/home/pclint/lint.log", append=true)
```

`-zero | -zero(#)` sets exit code to 0

This is useful to prohibit the premature termination of `make` files.

`-zero(#)` will set the exit code to zero if all reported errors are numbered `#` or higher after subtracting off 1000 if necessary. More precisely, messages that have a message number whose modulus 1000 is equal to or greater than `#`, do not increment the error count reported by the exit code. Note that suppressed errors also have no effect on the exit code. Use this option if you want to see warnings but proceed anyway. This option has no effect if the `frz` flag is ON.

`+zero_err(# [#...])` specify message numbers that should not increment exit code
`-zero_err(# [#...])` specify message numbers that should increment exit code

These options allow fine-grained control over exactly which messages contribute to the return value. Additionally, the `-exitcode` option allows the return value to be explicitly and unconditionally set.

When the `frz` is OFF, the return value from PC-lint is the number of messages emitted. The `-zero_err` option can be used to remove message numbers from this set and `+zero_err` can be used to add messages from this set. For example,

```
-zero\_err(*)
```

```
+zero\_err(w1)
+zero(80??)
```

will cause the exit code to be set to the number of error messages plus the number of messages issued in the 8000-8099 range. This option has no effect if the `frz` flag is ON.

4.6.3 Output

`-env_pop` pop the current option environment
`-env_push` push the current option environment
`-env_restore(`*Name*`)` restore the option environment to a previously saved one
`-env_save(`*Name*`)` save the current option environment with name *Name*

These options can be used to save and recall *option environments*. These options are similar to `-save` and `-restore` but operate in a larger context. Whereas `-save` and `-restore` operate only on the base suppression set state (affected by `-e#`, `+e#`, `-w#`, and `+efreeze`/`-efreeze`), these (`-env_*`) options also affect parameterized suppressions (`-esym`, `-efunc`, `-estring`, etc.) as well as many other options, namely: formatting options, flag options, include directory options, append options, deprecate options, library options, and reserved word options. The option environment manipulated by these options does **not** include: strong type options, message group options, return value options, program info options, function semantic options, or pre-compiled header options. The options `-skip_function`, `-unit_check`, and `-subfile` are similarly not part of the option environment and not affected by these options.

The `-env_push` and `-env_pop` options are analogous to `-save` and `-restore`. The `-env_push` option saves the current state of the option environment and a corresponding `-env_pop` option restores that state. `-env_push` options can be nested.

The `-env_save(`*Name*`)` option stores away a snapshot of the current option environment and associates it with a *Name* that can be used to later recall the option environment with `-env_restore(`*Name*`)`. The *Name* may consist of letters, numbers, and the underscore and is case-sensitive.

Attempting to use `-env_pop` without first using `-env_push` or specifying a name for `-env_restore` that was not provided as a name to `-env_save` will result in error `72` (bad option).

`-oe(`*Filename*`)` redirect stderr to *Filename* overwriting existing content
`+oe(`*Filename*`)` redirect stderr to *Filename* in append mode

This is primarily used to capture the help screen. For example:

```
lint -oe(temp) +si4 ?
```

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.

`-os(`*Filename*`)` redirect stdout to *Filename* overwriting existing content
`+os(`*Filename*`)` redirect stdout to *Filename* in append mode

Causes output directed to standard out to be place in the file *Filename*. This is like redirection and has the following advantages: (a) the option can be placed in a `.lnt` file or anywhere that a lint option can be placed (b) not all systems support redirection and (c) redirection can have strange side effects. If `+os` is used rather than `-os`, output is appended to the file. Make sure this option is placed before the file being linted. Thus

```
lint -os(file.out) fil.c
```

is correct. But

```
lint fil.c -os(file.out)
```

loses the intended output. The reason is that the redirection doesn't start until the option is encountered.

`-stack(&file=`*`filename`*`, &overhead(`*`n`*`), &external(`*`n`*`), &off, name(`*`n`*`))` set stack reporting options
`+stack` enable stack reporting

The `+stack` version of this option can be used to trigger a stack usage report. The `-stack` version is used only to establish a set of options to be employed should a `+stack` option be given. To prevent surprises if a `-stack` option is given without arguments it is taken as equivalent to a `+stack` option. See Section 11.6 Stack Usage Report for more details and complete listing of the sub-options.

4.7 Special Detection Options

4.7.1 Strong Type

`-father(`*`parent`*`, `*`child`*` [,`*`child`*`...])` a stricter version of `-parent`

This option is like the `-parent()` option except that it makes the relationship a strict one such that a *child* type can be assigned to a *parent* type but not conversely. To make all relationships strict you may use the `-fhd` option. (Turn off the Hierarchy Down flag). If a `-parent()` option and a `-father()` option are both given between the same two types then the relationship is considered strict. See Section 7.5.4 Restricting Down Assignments (-father)

`-index(`*`flags`*`, `*`ixtype`*`, `*`sitype`*` [,`*`sitype`*`...])` establish *ixtype* as index type

This option is supplementary to and can be used in conjunction with the `-strong` option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype* (or *sitype*'s if more than one is provided). Both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a `typedef` declaration. See Section 7.3 Strong Types for Array Indices

`-parent(`*`parent`*`, `*`child`*` [,`*`child`*`...])` augment strong type hierarchy

This option adds a link or links to the strong type hierarchy. See Section 7.5.3 Adding to the Natural Hierarchy

`-strong(`*`flags`*` [,`*`name`*`...])` imbues typedefs with strong type checking characteristics

Identifies each *name* as a strong type with properties specified by *flags*. Presumably there is a later `typedef` defining any such *name* to be a type. Strong types are completely described in Chapter 7 Strong Types.

4.7.2 Miscellaneous Detection

`-deprecate(`*`Category`*`, `*`Name`*` [,`*`Commentary`*`])` deprecates the use of *Name* within *Category*

The `-deprecate` option can be used to mark variables, macros, functions, keywords, types, options, and conversion specifiers as deprecated. When a deprecated entity is used in source code, message 586 will be issued warning of the use of a deprecated entity. See 11.8 Deprecation of Entities for more information.

`-idlen(`*`count`* `[,`*`options`*`])` specifies the number of meaningful characters in identifier names

This option defines the number of significant characters for different types of identifiers, as well as whether those names should be treated as case-sensitive, for the purpose of reporting name clashes in C modules. When this option is used with a value of *`count`* greater than zero, PC-lint Plus will report on pairs of identifiers in the same name space that are identical in their first *`count`* characters but otherwise different. *`Options`* are:

Option	Meaning
x	eXternal symbols
p	Preprocessor symbols
c	Compiler symbols

If omitted, all symbols are assumed. Uppercase versions of these options may be used to additionally specify that symbol names are case insensitive.

The C language Standards define minimum limits on the number of significant characters of an identifier. Compilers or linkers that employ such a limit will ignore all but the first *`count`* characters. The `-idlen` option can be used to find pairs of identifiers that are identical in the first *`count`* characters but are nonetheless different. PC-lint Plus treats the identifiers as different but reports on the clash.

The minimum identifier limits defined by C and the corresponding `-idlen` values are show below.

Language Version	External Identifier Minimum	Internal Identifier Minimum	Preprocessor Identifier Minimum	PC-lint Plus options
C89	6 case-insensitive characters	31 case-sensitive characters	31 case-sensitive characters	`-idlen(6,X)` `-idlen(31,pc)`
C99	31 case-sensitive characters	63 case-sensitive characters	63 case-sensitive characters	`-idlen(31,x)` `-idlen(63,pc)`
C11	31 case-sensitive characters	63 case-sensitive characters	63 case-sensitive characters	`-idlen(31,x)` `-idlen(63,pc)`

Option `x`, *external* symbols, refers to functions and variables with external linkage. Option `p`, *preprocessor* symbols, refers to macros and parameters of function-like macros. Option `c`, *compiler* symbols, refers to all the other symbols and includes symbols local to a function, struct/ union tags and member names, enum constants, etc.

Message 621 is issued when two *external* identifiers clash (across the entire program, regardless of scope), when two *compiler* identifiers or a *compiler* and an *external* identifier clash in the same name space, or when *preprocessor* identifiers clash. The description of message 621 lists all of the different cases where clashes are reported.

Message 621 may be suppressed for individual identifiers or types of clashes using the `-estring` option. `-idlen` is off (equivalent to `-idlen(0)`) by default.

`+misra_interpret(`*`Language*Year, interpretation`*`)` enable MISRA interpertation
`-misra_interpret(`*`Language*Year, interpretation`*`)` disable MISRA interpertation

These options allow for the interpretation of MISRA standards to be configured. The first argument is the language and year of a MISRA standard, one of `c2012`, `c++2008`, or `c2004`. The second argument is a string which determines which interpretation is modified. These strings are descriptions of the behavior that will change. There are no shorter variants or abbreviations. Ensure you follow any necessary documentation process for your project when modifying the interpretation of MISRA rules.

The configurable interpretations are:

- `c2012`
  `essential type differs from standard type only for int and unsigned int`
  (default ON)
  When enabled, the essential type of an expression will simply be the standard type of the expression unless the standard type of the expression is (`signed`) `int` or `unsigned int`. Otherwise, such restrictions will only apply when specifically called out by normative text. For example, the essential type of `5UL * 5UL` will be `unsigned long` when this is ON and `unsigned char` when it is OFF.
- `c2012`
  `essential type of sizeof is the UTLR of the result`
  (default OFF)
  When enabled, the essential type of `sizeof` and `_Alignof` expressions that yield constant values will be the unsigned type of lowest rank for the value.
- `c++2008`
  `underlying type of explicit cast is the cast type`
  (default ON)
  When enabled, the underlying type of an explicit cast will be the type that the subexpression was cast to even when the entire expression is an integer constant expression. Otherwise, the entire expression, including the explicit cast, will be be assigned an underlying type based on the resulting constant value.
- `c++2008`
  `permit unmodified non-const incomplete array parameters`
  (default OFF)
  MISRA C++ requires that variables be declared using the `const` qualifier unless they are modified. MISRA C++ also requires that a pointer parameter to which array indexing is applied be declared using array syntax. It not possible to declare a `const` pointer parameter using array syntax. When enabled, this impossible combination of requirements is relaxed by permitting the lack of const on a parameter of pointer type that was declared as an incomplete (unsized) array.
- `c2004`
  `allow types of equal size for rule 10.1`
  (default ON)
  When enabled, violations of rule 10.1 will not be reported for "conversions" between identical but distinct integer types (for examle, `int` and `long` could potentially be the same size). Note that even if this is disabled, the word "wider" in rule 10.1 (a) specifically will be interpreted as "not smaller" as required for the rule's own examples.

4.7.3 Semantic

`-function(`*`Function0`*` [,`*`Function1`*`] [,`*`Function2...`*`])` copy or remove semantics from *Function0*

This option specifies that *Function1*, *Function2...* are like *Function0* in that they exhibit special properties normally associated with *Function0*. The special functions with built-in meaning are described in Section 9.1 Function Mimicry (-function). See also `-sem` in Section 9.2 Semantics Specifications.

`-printf(`*N*`, `*name1* `[,`*name2*`...])` specified *name*s are `printf`-like functions with format provided in the *N*th argument

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

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

For such functions, the types and sizes of arguments starting with the variant portion of the argument list are expected to agree in size and type specified by the format. The variant portion of the argument list begins where the ellipsis is given in the function declaration. See also `-scanf`, below and Sections 9.1 Function Mimicry (-function) and 11.1 Format Checking.

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)
```

`-scanf(`*N*`, `*name1* `[,`*name2*`...])` specified *name*s are `scanf`-like functions with format provided in the *N*th argument

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

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

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

`-sem(`*Function* `[,`*Sem*`...])` associates the semantic *Sem* with *Function*

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

To represent this semantic you could have used the option:

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

The details of semantic specifications are contained in Section 9.2 Semantic Specifications.

4.7.4 Value Tracking

`-specific_climit` maximum number of specific calls per function

The total number of specific calls recorded for any one function is limited to *n*. Because of recursion, the total number of specific calls made on any one function can be huge. This option prevents any one function from hogging resources. By default, the value is 0, implying no limit.

`-vt_depth=`*n* specifies the maximum number of nested specific walks

The maximum call stack depth for specific walks during value tracking. This limits the number of nested specific walks and also acts as an upward limit on recursion depth.

`-vt_passes=`*n* specifies the number of passes for intermodule value tracking

The number of times the entire set of project modules will be rescanned to perform additional intermodule value tracking using specific calls saved from the previous pass. Each module is processed during the initial pass where intramodule messages are issued and again during global wrap-up (if it has not been disabled), and each of these constitute a pass. This option can be used to request additional auxiliary passes beyond those that arise naturally from the processing architecture.

4.8 Meta Characters for Options

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

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

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

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

For example

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

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

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

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

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

suppresses any message about symbol `name`.

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

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

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

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

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

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

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

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

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

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

```
//lint -etype(1502, "B<float, int>")
// Without the double quotes, -esym() sees three arguments:
// "1502", "B<float", and "int>"
template <class T, class U> class B { };
B<float, int> b;
```

Both wildcard characters and non-wildcard characters may be used both within and outside of the double-quoted sequence. A right parenthesis or comma that appears outside of a double-quoted sequence marks the end of the argument as usual.

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

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

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

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

4.9 How Suppression Options are Applied

PC-lint Plus has a rich set of options that govern when and how diagnostics are issued. It is possible for multiple options to affect the issuance of a particular diagnostic. This section describes the process by which PC-lint makes the determination to issue a specific diagnostic, including the interaction of multiple relevant suppression options.

Note that the term "suppression options" is used here to refer both to options that suppress a message (e.g. `-e#`) and options that enable a message (e.g. `+esym`).

PC-lint Plus performs the following steps, in the order given, to determine if the message should be suppressed or issued.

1. If the message is an unsuppressible message (errors 305, 309, 315, 330, and 367), the message is issued.

2. If a single-line suppression exists on the same line as the location given in the message, and the message is not frozen (with the `+efreeze` or `++efreeze` options) at the point of the suppression, the message is suppressed.

3. If the location of the message is subject to a scoped suppression (`-e(#)`, `--e(#)`, `-e#`, or `--e#`, including those resulting from `-emacro((#))`, etc. options), and the message is not frozen at the point of suppression, the message is suppressed.

4. A voting mechanism is next employed to determine whether to issue the message. If the message is currently in the set of enabled messages (this is the message set that is manipulated by the `-w`/`-e`/`+e` options), one vote is cast in favor of issuing the message. Each `-esym`, `-etype`, `-estring`, `-egrep`, `-efile`, `-ecall`, `-elibcall`, `-elibsym`, `-elibmacro`, and `-emacro` option that applies to the current message casts its votes† in favor of suppressing the message for each parameter in the message that is matched by the option. Similarly, each applicable `+esym`, `+etype`, `+estring`, `+egrep`, `+efile`, `+emacro`, `+ecall`, and `+elibcall` option casts its votes in favor of emitting the message. If the number of votes to suppress is greater than or equal to the number of votes to issue the message, the message is suppressed.

5. If the message location refers to a library region, and the message is suppressed for libraries via the `-elib` or `-wlib` options, the message is suppressed.

6. The message is issued.

† Parameterized suppression options cast one vote for each parameter in the message that matches the provided pattern.

Example
Elective Note 9001 (octal constant '*String*' used) is parameterized by a single string (the octal constant encountered). Given a file `example.c` that contains:

```
int i1 = 00;
int i2 = 01;
```

If PC-lint Plus is run with the options `-w1 +e9001` (disabling all non-error messages and then enabling note 9001), the messages issued are:

```
note 9001: octal constant '00' used
note 9001: octal constant '01' used
```

The `+e9001` results in one vote in favor of issuing the messages and nothing casts a vote against. If we add the option `-efile(9001, example.c)`, both messages will be suppressed because for each message the `+e9001` effects one vote in favor of issuing and the `-efile` option votes against issuing the note. Since there are not more votes to issue than to suppress, the message is suppressed. If we also add the option `+estring(9001, 00)` an extra vote will be cast to issue the message when the message references the octal constant `00` resulting in the output:

```
note 9001: octal constant '00' used
```

In other words, when using the options `+e9001 -efile(9001, example.c) +estring(9001, 00)`, message 9001 will be allowed in all files except `example.c` where only uses of the octal constant `00` will be reported.

4.10 Flag Options

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

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

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

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

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

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

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

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

Flag	Default	Summary	Version
f12	OFF	view MISRA C 2012 essential types	1.0
f@m	OFF	commerical '@' is a modifier	—
fac	OFF	allow instantiation of abstract classes	1.0
fai	ON	arguments pointed to get initialized	—
fan	OFF	support anonymous unions	—
fas	OFF	support anonymous structs	—
fat	ON	parse .net attributes	—
fau	OFF	bitwise AND with negative constant is unknown	1.2
fba	OFF	bit addressability	—
fbe	OFF	enable backslash escapes for special option characters	1.1
fbl	OFF	dependent base class lookup in templates	1.0
fbo	ON	activate `bool`, `true`, `false`	—
fca	ON	convert attributes to semantics	1.0
fcc	OFF	capitalize message categories	1.0
fce	OFF	continue on `#error`	—
fcm	ON	copy semantics from macro definitions	1.0
fcn	ON	convert non-printable characters in context line	1.0
fcp	OFF	all subsequent modules are considered C++	—
fcs	OFF	continue on static assertion failure	1.0
fcu	OFF	`char` is unsigned	—
fcw	ON	attribute responsibility for last write in callee to caller	1.2
fdd	ON	dimensional by default	—
fdg	ON	expansion of digraphs	—
fdh	OFF	append '`.h`' to header names in `#include`'s	—
fdi	ON	search directory of including file	—
fdl	OFF	pointer difference is `long`	—
fdm	OFF	comma from macro expansion does not delimit macro args	1.0
fdt	OFF	delayed template parsing	1.0
fdu	ON	allow '-d'/'-u' options in lint comments	1.0
fdx	OFF	consider use of operator delete to be a modification	1.0
fee	ON	expand environment variables	1.0
fei	OFF	underlying type for `enum` is always `int`	—
fes	OFF	search enclosing scopes for friend tag decls	1.0
fet	OFF	require explicit throw specifications	—
ffb	ON	for loop creates separate block	—
ffc	ON	non-library functions assume custody of non-const pointers	—
fff	OFF	fold filenames to a consistent case	—
ffn	OFF	use full file names	—

Flag	Default	Summary	Version
ffv	OFF	implicit function to void pointer conversion	1.0
ffw	OFF	allow friend decl to act as forward decl	1.0
fgi	OFF	`inline` treated as GNU inline	1.0
fgl	ON	use GNU line markers in preprocessed output	1.2
fhd	ON	allow hierarchy downcasts	—
fho	OFF	header include guard optimization	1.0
fhs	ON	natural hierarchy of strong types	—
fhx	ON	hierarchy of index types	—
fia	OFF	inhibit supplementary messages	1.0
fie	OFF	use the integer model for enums	—
fim	ON	`-i` can have multiple directories	—
fin	OFF	refer to supplemental messages with the info label	1.0
fiw	ON	initialization is a write	—
fiz	ON	initialization by zero is a write	—
fkp	OFF	use K&R preprocessor	—
fla	ON	locations for all diagnostics	1.0
flb	OFF	treat code as library	—
flf	OFF	process library functions	—
fll	OFF	allow `long long int`	—
flm	OFF	lock message format	—
fln	ON	honor `#line` directives for diagnostics	—
flp	OFF	lax null pointer constants	1.0
fma	OFF	microsoft inline `asm` blocks	1.0
fms	OFF	microsoft semantics	1.0
fmt	OFF	match template template-arguments to compatible templates	1.2
fmx	ON	enable member access control in C++	1.2
fnc	OFF	nested comments	—
fnf	OFF	fall back to operator `new` when `new[]` not available	1.0
fnn	OFF	new can return null	—
fnr	OFF	null pointer return	—
fon	ON	support for C++ operator name keywords	—
fpa	OFF	pause before exiting	—
fpe	ON	use precision of enumerators instead of explicit enum base type	1.2
fpm	OFF	limit precision to the maximum of the arguments	—
fpn	OFF	pointer parameter may be null	—
fpo	ON	limit precision to the type of the operation	—
fqb	ON	qualifiers go before types	—
frc	OFF	remove commas before `__VA_ARGS__`	1.0
frd	OFF	redefine default params for class template function members	1.0
frz	ON	use return code only to indicate execution failure	1.0
fsc	OFF	strings are `const char*` even in C	—
fsd	OFF	output stack diagrams	1.0
fse	OFF	use smallest underlying type for enums	1.0
fsf	OFF	display function names for semantics during calls	1.0
fsi	OFF	search `#include` stack	1.0
fsl	OFF	single line comments	1.0
fsn	ON	treat strings as names	—
fso	OFF	return semantics override deduced return values	1.0

Flag	Default	Summary	Version
fsp	ON	specific calls	—
fsv	ON	track static variables	—
fta	ON	enable typographical ambiguity checks	—
ftg	ON	permit trigraphs	—
fum	OFF	user declared move deletes only corresponding copy	1.0
fun	OFF	issue additional stack usage notes	1.0
fur	OFF	allow unions to contain reference members	1.0
fvd	OFF	interactive value tracking debugger	1.0
fwu	OFF	`wchar_t` is unsigned	—
fzd	OFF	enable sized deallocations	1.2
fzl	OFF	`sizeof` is `long`	—
fzu	ON	`sizeof` is unsigned	—

`f12` view MISRA C 2012 essential types (default OFF).

Issues message 9903 after full expressions to illustrate the MISRA C 2012 essential types involved in the expression and how they combine to form new essential types.

For example, given:

```
int f(int a, short b) {
    return (b + b) + (a + b);
}
```

PC-lint Plus emits:

```
note 9032: left operand to + is a composite expression of type
    'signed16' which is smaller than the right operand of type 'signed32'
    return(b + b) + (a + b);
           ~~~~~~ ^
```

Why is the left side `signed16` and the right side `signed32`?

Processing the example with `+f12` and `+e9903` yields the step by step evaluation of the expression with the corresponding essential types involved at each step:

```
info 9903: (signed16 + signed16) + (signed32 + signed16)
info 9903: (signed16) + (signed32)
info 9903: signed32
```

`f@m` commerical '`@`' is a modifier (default OFF).

This is a feature required by some embedded compilers that employ a syntax such as:

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

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

`fac` allow instantiation of abstract classes (default OFF).

Some compilers allow instantiation of abstract classes (e.g. Visual C++ 6). If this flag is ON, such instantiations will be allowed, otherwise an error will be emitted and the class will not be instantiated.

`fai` arguments pointed to get initialized (default ON).

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 = 0;
    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` support anonymous unions (default OFF).

If this flag is ON, anonymous unions are supported in C90 and C99 modes (they were added to the language in C11). Anonymous unions appear within structures and have no name within the structure so that they must be accessed using an abbreviated notation. For example:

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

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

`fas` support anonymous structs (default OFF).

If this flag is ON, anonymous structs are supported in C90 and C99 modes (they were added to the language in C11) as well as in C++. Anonymous structs are similar to anonymous unions. That is, if a struct has no tag and no declarator as in:

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

then references to the members of the inner struct are as if they are members of the containing struct. Thus `x.a` refers to member a within the unnamed struct within `struct X`.

`fat` parse .net attributes (default ON).

Dot net (`.net`) attributes are contained within square brackets. E.g.

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

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

`fau` bitwise AND with negative constant is unknown (default OFF).

If this flag is ON, Value Tracking will treat the result of a bitwise AND operation between an unknown value and constant value as unknown. If it is OFF, the result will be based on the value of the constant and the range of the type of the unknown operand.

`fba` bit addressability (default OFF).

If this flag is ON (by default it is OFF), an individual bit of an int (or any integral) can be specified using the notation:

a.integer-constant

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

For example, the following code:

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

will set the 2nd bit of `n` to 1 and the 4th bit to 0. Later it tests those bits. This syntax is not standard C/C++ but does represent a common convention used by compilers for embedded systems.

`fbe` enable backslash escapes for special option characters (default OFF).

When the `fbe` flag is enabled, the backslash character can be placed immediately before any of the following special characters to remove the special meaning of that character:

```
{ } ( ) [ ] ! , " \
```

When this flag is enabled, any other character following a backslash inside an option will be met with eror 72 (bad option). The `fbe` flag can be turned ON and OFF between options.

`fbl` dependent base class lookup in templates (default OFF).

When this flag is ON, unqualified lookup in a class template will result in a search of dependent base classes. This is non-standard behavior implemented by some compilers.

`fbo` activate `bool`, `true`, `false` (default ON).

If this flag is ON, keywords `bool`, `true`, and `false` are activated at the start of every C++ module.

`fca` convert attributes to semantics (default ON).

If this flag is ON, certain attributes appearing in function declarations using the GCC attribute syntax will automatically be converted to function semantics. The supported attribute to semantic mappings are:

Attributer	**Equivalent**
`format(printf, index, first-to-check)`	`-printf(index, func)`
`format(scanf, index, first-to-check)`	`-scanf(index, func)`
`noreturn`	`-sem(func, r_no)`
`nonnull(i)`	`-sem(func, iP)`
`const`	`-sem(func, pure)`
`pure`	`-sem(func, pure)`

For example, given the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
              __attribute__ ((format (printf, 2, 3)));
```

the effect of the option `-printf(2, my_printf)` is automatically applied to the function. Semantics applied from attributes only affect the overload in which the attribute appears.

`fcc` capitalize message categories (default OFF).

In PC-lint, the message categories (Error, Warning, Info and Note) were spelled with an initial uppercase letter. In PC-lint Plus, these categories are presented in all lower case. Setting this flag will emulate the PC-lint 9 behavior. If the value of the flag is 2 or greater (using `++fcc`, the message categories will be presented in all upper case (ERROR, WARNING, etc).

`fce` continue on `#error` (default OFF).

PC-lint Plus will normally terminate with a fatal error (309) when a `#error` preprocessing directive is encountered. If this flag is set to a value of 2 or greater, PC-lint Plus will still emit the error message but continue processing. This may be useful to help troubleshoot incomplete configurations in a handful of circumstances but this option should never be employed in a production environment because its use will only serve to hide serious configuration issues.

When PC-lint Plus is forced to continue processing after encountering such an error, the generated AST may be incorrect/incomplete and resulting analysis can no longer be considered to be reliable. Because of this, message 686 will be issued when this option is used. Additionally, note that while processing will continue when using this option, message 309 is still emitted and cannot be suppressed. Message 309 typically indicates missing macro definitions and can usually be resolved by examining the context in which the error is emanating and defining the appropriate macro(s).

`fcm` copy semantics from macro definitions (default ON).

This flag controls when function semantics are copied from one function to another due to a macro definition. For example, if the following macro definition is encountered:

```
#define malloc xmalloc
```

calls to `malloc` would be changed to calls to `xmalloc` but the built-in function semantics for the `malloc` function would not be copied to `xmalloc`. The same issue exists for functions with user-defined function semantics.

When this flag is ON (default), semantics are copied only if the name of the new function is the same as the old function with optional underscores at the beginning and/or end of the new name and message 828 will be issued to announce the semantic copying. In all other cases, semantics will not be copied and warning 683 will be issued.

When this flag is OFF (`-fcm`), semantics are never copied and message 683 will always be issued when a `#define` is used to define a macro with the same name as a function that has semantics.

If this flag is set to a value of 2 (`++fcm`), semantics are always copied, producing message 828 each time.

Note that if the new name corresponds to a function that already has semantics associated with it, no copying is performed and no message is issued.

`fcn` convert non-printable characters in context line (default ON).

This flag controls how non-printable characters in source code lines are represented in the context line of messages. When this flag is ON, non-printable characters are presented using the syntax `<U+dddd >` where `dddd` is the hexadecimal value of the UTF32 version of the character. Note that this conversion only occurs for the context lines included in emitted messages. If the flag is OFF the character is printed as-is, without conversion.

Note: If you are using a message height of 2 (which causes the position indicator to be embedded in the context line), and your position indicator includes non-printable characters (such as ANSI/VT100 terminal escape sequences), you will need to turn this flag OFF to keep those characters from being converted.

`fcp` all subsequent modules are considered C++ (default OFF).

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

`fcs` continue on static assertion failure (default OFF).

PC-lint Plus will normally terminate with a fatal error (330) when a static assertion fails. If this flag is set to a value of 2 or greater, PC-lint Plus will still emit the error message but continue processing. This may be useful to help troubleshoot incomplete configurations in a handful of circumstances but this option should never be employed in a production environment because its use will only serve to hide serious configuration issues.

When PC-lint Plus is forced to continue processing after encountering such an error, the generated AST may be incorrect/incomplete and resulting analysis can no longer be considered to be reliable. Because of this, message 686 will be issued when this option is used. Additionally, note that while processing will continue when using this option, message `330` is still emitted and cannot be suppressed. Message `330` typically indicates missing or incorrect scalar type sizes (which can be configured with the `-s` option), missing type definitions, or missing or incorrectly specified macros. Review the message details and the context in which the failure occurs to address the underlying cause.

`fcu` `char` is unsigned (default OFF).

If this flag is ON, plain `char` declarations are assumed to be `unsigned`. This is useful for compilers that, by default, treat `char` as `unsigned`. Note that this treatment is specifically allowed by the ANSI/ISO standard. That is, whether `char` is `unsigned` or `signed` is up to the implementation.

`fcw` attribute responsibility for last write in callee to caller (default ON).
`fdd` dimensional by default (default ON).

If this flag is ON then strong types (with the 'J' flag) will be considered equivalent to 'Jd'. The resulting behavior will be equivalent to treating the type as a physical dimension such as meters or seconds. See Section 7.4 Dimensional Analysis.

`fdg` expansion of digraphs (default ON).

By default, PC-lint Plus will expand digraph sequences. If this flag is turned off, digraph sequences will not be expanded.

`fdh` append '`.h`' to header names in `#include`'s (default OFF).

When the Dot-H flag is ON (`+fdh`) and if an extension-less header is seen, an attempt is made to open the file first with the `.h` extension and, that failing, open the file using the original name. If the flag has a value of 2 or higher, an attempt is made to open the `.h` extended name but not on the original name.

`fdi` search directory of including file (default ON).

If this flag is ON, the search for `#include` files will start with the directory of the including file (in the double quote case) rather than with the current directory. This is the standard Unix convention and is also used by the Microsoft compiler. For example:

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

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

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

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

`fdl` pointer difference is `long` (default OFF).

This flag specifies that the difference between two pointers is typed `long`. Otherwise the difference is typed `int`. This flag is automatically adjusted upon encountering a `typedef` for `ptrdiff_t`.

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

```
+fdl ++fdl
```

`fdm` comma from macro expansion does not delimit macro args (default OFF).

When this flag is ON, single commas from nested macro expansions are not treated as argument separators. In PC-lint 9 this behavior was implemented via the option.

```
+compiler(comma_from_macro_expansion_does_not_delimit_macro_args)
```

`fdt` delayed template parsing (default OFF).

When this flag is ON, parsing of function template definitions will occur at the end of the module instead of when the definition is initially encountered. This is necessary to properly parse certain constructs that are not technically valid C++ but are allowed (and employed) by some implementations.

`fdu` allow '-d'/'-u' options in lint comments (default ON).

The `fdu` flag controls whether `-d`/`+d` and `-u` options have an effect inside of lint comments. When this flag is ON, these options are honored when they appear inside of lint comments. When this flag is OFF, such options are not honored and a 686 message will be issued.

Modules with large numbers of macro definitions may take noticeably longer to process when this flag is ON. Turning this flag OFF in such situations can improve performance.

`fdx` consider use of operator delete to be a modification (default OFF).

When this flag is ON, the application of `operator delete` will make its argument ineligible for the suggestion that it could be a pointer to const. While it is legal to delete a pointer to const, this can subvert the common expectation that the target of a pointer to const will not be changed.

`fee` expand environment variables (default ON).

This flag controls how environment variables are handled when appearing in lint options surrounded by percent signs, e.g. `%PATH%`. If this flag is OFF, environment variables are not expanded. If this flag is set to a value of 1 (the default), environment variables are expanded but not recursively. If this flag is set to a value greater than 1, environment variables are recursively expanded.

`fei` underlying type for `enum` is always `int` (default OFF).

If this flag is ON, the underlying type of enumerations will always be '`int`'. Otherwise, the Standard C/C++ rules will be used for determining the underlying type.

`fes` search enclosing scopes for friend tag decls (default OFF).

If this flag is ON, name lookup will consider enclosing scopes for unqualified friend tag declarations that are not template-ids, allowing redeclaration from an enclosing namespace. In Standard C++, only scopes within the innermost enclosing namespace are considered. Note that this lookup scope extension occurs only for types, not functions.

`fet` require explicit throw specifications (default OFF).

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

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

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

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

The disadvantage, however, is that by adding an exception specification you are saying that the function throws no exception other than those listed. If a library function throws an undeclared exception (such as `abs()` above) you will get the dreaded `unexpected()` function call. See [1, Item 14], Scott Meyers "More Effective C++".

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

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

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

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

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

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

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

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

`ffb` for loop creates separate block (default ON).

The C++ standard designates that variables declared within `for` clauses are not visible outside the scope of the `for` loop. For example, in the following code, `i` cannot be used outside the `for` loop.

```
for( int i = 0; i < 10; i++ ) {
    // ...
}
// can't use i here.
```

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

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

`ffc` non-library functions assume custody of non-const pointers (default ON).

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

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

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

`fff` fold filenames to a consistent case (default OFF).

If this flag is ON, file names are processed case-insensitively such that `X.C` may refer to a file with the name `x.c`, `#include "a.h"` can be used to include a header with the name `A.H`, etc. The options `+lnt`, `+cpp`, and `+ext` are also effected, e.g. after `+cpp(cc)` both ".cc" and ".CC will be considered to be C++ extensions. This flag is only intended for use with case-insensitive filesystems.

`ffn` use full file names (default OFF).

When this flag is ON filenames reported in error messages are full path names. This can assist editors in locating the correct position within files when default directories may be different than during the linting process. If this flag is OFF (default), filenames are reported as provided to PC-lint Plus. If this flag has a negative value (`--ffn`), the filenames are reported using just the base file name.

`ffv` implicit function to void pointer conversion (default OFF).

If this flag is ON, implicit "function pointer" to "`void` pointer" conversions are allowed in C++ mode (they are always allowed in C mode).

`ffw` allow friend decl to act as forward decl (default OFF).

When this flag is ON set, friend declarations act as forward declarations. This is not standard C++ behavior but is supported by some compilers.

`fgi` `inline` treated as GNU inline (default OFF).

When this flag is ON, GNU inline semantics are applied to entities declared with the `inline` keyword. Namely, declarations with `inline` that are not declared `static` are externally visible even if no `extern` specifier is present. This matches the behavior of GNU90 mode.

`fgl` use GNU line markers in preprocessed output (default ON).
`fhd` allow hierarchy downcasts (default ON).

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

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

Then an assignment from an object of type `X` to a variable of type `Y` will draw a warning if the flag is turned off. See Sections 7.5.4 Restricting Down Assignments (-father) and 7.5.3 Adding to the Natural Hierarchy.

`fho` header include guard optimization (default OFF).

This flag controls the handling of header files that utilize include guards. If this flag is ON, header files with valid include guards are not re-processed and as such they will not show up in verbosity messages (with `-vi` or `-va`) and lint options that appear before the include guard or in files included after the include guard will not be executed after the initial inclusion. If the flag is OFF, files will be entered every time they are referenced. Note that in neither case are the contents of the guarded region re-processed, this flag controls only whether or not the file is re-entered before making the decision to re-process. This flag does not affect headers using `#pragma once`, such files are never re-entered regardless of the value of this flag.

`fhs` natural hierarchy of strong types (default ON).

If this flag is ON (it is by default) strong types are considered to form a hierarchy based on `typedef` statements. See Section 7.5.2 The Natural Type Hierarchy and Section 7.5.3 Adding to the Natural Hierarchy.

`fhx` hierarchy of index types (default ON).

If this flag is ON (it is by default) strong index types are related via the type hierarchy. See Chapter 7 Strong Types. See also the `+fhs` flag.

`fia` inhibit supplementary messages (default OFF).

If this flag is ON, supplementary messages (`831` and `890-899`) will be suppressed.

`fie` use the integer model for enums (default OFF).

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

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

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

`fim` `-i` can have multiple directories (default ON).

With this flag ON, the `-i` option may specify multiple include directories (like the INCLUDE environment variable). For example,

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

will have the same effect as:

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

`fin` refer to supplemental messages with the info label (default OFF).

By default, supplemental messages are labeled as supplemental. In PC-lint, such messages were labeled as info. If this flag is ON, the PC-lint behavior is used.

`fiw` initialization is a write (default ON).

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

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

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

`fiz` initialization by zero is a write (default ON).

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

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

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

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

`fkp` use K&R preprocessor (default OFF).

ANSI/ISO C provides several facilities of the preprocessor that were not part of K&R C including the `#elif` directive and the `defined` keyword. If this flag is ON, use of these constructs will be warned about via messages `555` and `517`, respectively.

`fla` locations for all diagnostics (default ON).

If this flag is ON, filename information will be provided even when the location does not correspond with a physical source file. In such cases, a representative filename that describes the location will be provided in angle brackets. For example, a location on the command line will be referenced as `"<command line>"`, a location that represents a temporary buffer such as a macro expansion or string literal concatenation will be referenced as `"<scratch space>"`, a location within the LINT environment variable will be referenced as `"<LINT var>`, etc. If this flag is OFF, the filename information will be empty in such cases.

`flb` treat code as library (default OFF).

If ON, code is treated as belonging to a library header (See Section 5.1 Library Header Files). That is, declared objects do not have to be used or defined and messages specified by `-elib` are suppressed. This flag has been largely superseded by the notion of "Library Header Files" (See Section 5.1 Library Header Files). It still has its uses though. For example, the output of PC-lint Plus in preprocess mode (i.e. using the `-p` option) will contain Lint comments bearing `++flb` before and `--flb` after the positions at which library headers were included.

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

`flf` process library functions (default OFF).

If this flag is OFF, non-dependent library function definitions will not be fully processed. This saves some time and avoids issues stemming from library headers making use of intrinsic functions unknown to PC-lint Plus. If this flag is ON, all library function definitions will be fully processed. When set to a value of 1 (`+flf`), library functions will not be walked during value tracking analysis. To enable value tracking within library functions, set this flag to a value of 2 (`+flf ++flf`). Setting this flag to a negative value (`--flf`) will disable processing of all library function bodies, including dependent functions.

See also the `-skip_function` option, which can be used to skip processing on a per-function basis.

`fll` allow `long long int` (default OFF).

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

`flm` lock message format (default OFF).

This flag can be used by GUI front ends that depend on a particular format for error messages. If this flag is ON, the Message Presentation Options (See Section 4.3.3 Message Presentation) are frozen. That is, subsequent `-h`, `-width`, and the various `-format` options are ignored. Also ignored is the `-os` option. The `-os` option (See Section 4.6.3 Output) designates which file will receive error messages.

`fln` honor `#line` directives for diagnostics (default ON).

By default, `#line` directives affect the location information within error messages. The option `-fln` may be used to ignore `#line` directives. See Section 12.3 #line and #.

`flp` lax null pointer constants (default OFF).

In C++98, a null pointer constant was defined as an integral constant expression that evaluates to zero. Post C++11 the definition was changed to "an integer literal with value zero or a prvalue of type `std::nullptr_t`". When in C++11 and later modes, an expression such as `1 - 1` or `"\0"` will therefore not be considered to be a null pointer constant by default. If this flag is ON then the more lax C++98 semantics will be applied for such expressions.

`fma` microsoft inline `asm` blocks (default OFF).

This flag enables parsing support for Microsoft ASM blocks.

`fms` microsoft semantics (default OFF).

Setting this flag enables a number of Microsoft specific extensions that are not of interest to other compilers and therefore do not have their own options. This flag also enables a number of undocumented features and emulates several MS-specific bugs including:

- type definition in anonymous struct or union
- pure specification on function definition defined at class scope
- `sealed`, `override`, and `__except` contextual keywords
- explicit specializations within class scope
- forward references to `enum` types
- flexible array member in unions and empty classes
- support for `throw(...)` specification

`fmt` match template template-arguments to compatible templates (default OFF).
`fmx` enable member access control in C++ (default ON).
`fnc` nested comments (default OFF).

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

`fnf` fall back to operator `new` when `new[]` not available (default OFF).

If this flag is enabled and a placement `new[]` is called at a point where no valid array placement `new` declaration exists, instead of giving up, PC-lint Plus will try to "fall back" to a valid `operator new` function following Microsoft's behavior. For example, given:

```
void *operator new(size_t n, void *p, size_t limit);

int main() {
    char buffer[100];
    char *p = new(buffer, sizeof(buffer)) char[10];
}
```

PC-lint Plus will emit

```
error 1024: no matching function for call to 'operator new[]'
    char *p = new(buffer, sizeof(buffer)) char[10];
              ^ ~~~~~~~~~~~~~~~~~~~~~~~~~
```

since there is no valid `operator new[]` available. With the `fnf` flag enabled, the `operator new` function will be used instead and no error will be issued.

`fnn` new can return null (default OFF).

Turning this flag ON yields the old style `operator new`. That is, `new` may return NULL and does not throw an exception.

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

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

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

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

When this flag is OFF, using `std::nothrow` will still be considered to possibly return null. Decrementing this flag (from the default value of 0) will force `new` to never return null, even when it is explicitly requested that `new` not throw an exception.

`fnr` null pointer return (default OFF).

This flag is normally OFF. When this flag is ON, then all functions that return pointers and have no other return semantic are assumed to return pointers that could possibly be NULL. For example:

```
//lint +fnr     null can be returned by functions
int *f();
void g() {
```

```
        int *p = f();        // p could be NULL
        if (!p) return;      // avoid a diagnostic
        *p = 0;              // OK now to use p.
    }
```

`fon` support for C++ operator name keywords (default ON).

When this flag is ON (the default), the C++ alternative operator names:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, and xor_eq

are recognized as keywords with meaning equivalent to the operators:

`&&, &=, \&, |, ~, !, !=, ||, |=, ^,` **and** `^=`

respectively. If the value of this flag is 2 or higher, these operator names are also recognized in C mode (this is typically accomplished by `#including iso646.h` in C). If the flag is turned OFF, the alternative names are not recognized in C or C++.

`fpa` pause before exiting (default OFF).

When this flag is ON, PC-lint Plus will pause just before exiting (after all messages are produced), and request input through `stdin` after prompting on `stderr`. Hitting Return (i.e., Enter) should be enough to finally terminate. This option could be useful in a setup where PC-lint Plus is launched in a separate terminal window that closes when PC-lint Plus exits, before the desired output can be reviewed. This option should keep the window open. CAUTION: This option is recommended only as a trouble shooting option or as a stop gap measure. Some environments require the launched program to terminate or they themselves lock up.

`fpe` use precision of enumerators instead of explicit enum base type (default ON).

By default the precision of a value of `enum` type is based on the values of its enumerators. If this flag is turned OFF, C++11 enumerations defined with a fixed underlying type will use the precision of the specified type instead.

`fpm` limit precision to the maximum of the arguments (default OFF).

This is used to suppress certain kinds of Loss of Precision messages (734). In particular, if multiplication or left shifting is used in an expression involving `char` (or `short` where `short` is smaller than `int`) an unwanted loss of precision message may occur. For example, if `ch` is a `char` then:

```
ch = ch * ch
```

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

```
ch <<= 1
```

is not greeted with Message `734`.

`fpn` pointer parameter may be null (default OFF).

If this flag is set ON, all pointer parameters are assumed to be possibly NULL and a diagnostic will be issued if a pointer parameter is used without testing for NULL. For example:

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

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

For more information about this interesting test see Chapter 8 Value Tracking.

`fpo` limit precision to the type of the operation (default ON).

The precision of a mathematical operation is limited by the operation itself. For left shifting and right shifting, this flag is relevant only when the right hand operand is a known value. For left shifting, the resultant precision is first presumed to be the precision of the left hand operand plus the value of the right hand operand. For right shifting, the resultant precision is first presumed to be the precision of the left hand operand minus the value of the right hand operand. In both shifting cases, if the `fpm` flag is active, the precision is reduced to the smaller of that presumed precision and the precision of the left hand operand. If the `fpo` flag is active, the (possibly reduced) precision is reduced further still to the smaller of this reduced precision and the precision of the resultant type. For example:

```
void b(int c) {
    char d;
    d = c << 28;
}
```

will result in info 734, loss of precision, for the assignment regardless of whether or not this flag is active. When inactive, however, the precision reported in the message for the assigned value will be 59 bits and when active the precision will be 31 bits.

Likewise,

```
void b(int c) {
    if ((c << 28) == 2147483648)
        {}
}
```

will produce 650, `constant '2147483648' out of range for operator '=='`, when `fpo` is active and will not if the flag is inactive for the same reasons.

This flag also limits the precision for multiplication. Initially, the precision of the result is presumed to be the maximum precision of the operands. A tentative precision that equals the sum of the precision of the two operands is also calculated. If the higher ranked operand is signed and a sign is not possible in either of the operands, the tentative precision is reduced by one. If either operand is a power of two, the (possibly reduced) tentative precision of the result is reduced by one. If the `fpm` flag is inactive, the initial precision of the result will be discarded and replaced with this (possibly reduced) tentative precision. If the `fpo` flag is active the precision of the result will then be reduced to the smaller of the precision of the result so far calculated and the precision of the resultant type. For example:

```
void b(int c) {
```

```
        char d;
        d = c * 28;
    }
```

will result in information 734, loss of precision, for the assignment regardless of whether or not this flag is active. When inactive, however, the precision reported in the message for the assigned value will be 36 bits and when active the precision will be 31 bits.

Likewise,

```
    void b(int c) {
        if ((c << 28) == 34359738368)
            {}
    }
```

will exhibit the same 650 behavior as described above in the case of left shifting for the same reasons.

fqb qualifiers go before types (default ON).

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

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

[2] and [3] provide supporting evidence that not only is a convention useful, but that the better convention is the one rendered with `-fqb`.

frc remove commas before `__VA_ARGS__` (default OFF).

The variadic macro feature added in C99 has a commonly encountered limitation: it requires that at least one argument be provided for the variadic portion of the argument list. For example:

```
    #define LOG(format, ...) fprintf(stderr, format, __VA_ARGS__)
```

works fine when `LOG` is called with two or more arguments, e.g. `LOG("%s", "Started")`, but not when called with a single argument, e.g. `LOG("Started")`, which would expand to `fprintf(stderr, "Started",)` (note the trailing comma). As there is no facility provided by Standard C to address this limitation, various compilers have implemented their own extensions to deal with it.

GCC (and others) handle this by ascribing special meaning to the token pasting operator `##` when placed between a comma and a variable argument in a macro definition, causing the offending comma to be removed during expansion if the variable argument is left out or empty. This allows the above to be defined as:

```
    #define LOG(format, ...) fprintf(stderr, format, ##__VA_ARGS__)
```

and when invoked as `LOG("Started")` will expand to `fprintf(stderr, "Started")` (no trailing comma). PC-lint Plus supports this behavior regardless of the value of this flag but will issue message

2715 to alert of the non-portable behavior.

MSVC removes the trailing comma even without the appearance of the token pasting operator (as in the first example). This behavior is not enabled by PC-lint Plus by default, set this flag ON to enable support for this behavior.

`frd` redefine default params for class template function members (default OFF).

When this flag is ON, default parameters for member functions of a class template may be redefined and the new value will be ignored. This behavior is implemented by MSVC.

`frz` use return code only to indicate execution failure (default ON).

When this flag is ON, the exit code of PC-lint Plus will be 0 unless there was a fatal error. If this flag is OFF, the exit code will be the number of messages emitted, up to a max of 255 and can be manipulated with the `-zero`, `-zero_err`/`+zero_err`, and `-exitcode` options. If this flag is ON, the options that manipulate the exit code will have no effect.

`fsc` strings are `const char*` even in C (default OFF).

When this flag is ON, string constants are considered pointers to `const char`. For example:

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

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

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

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

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

`fsd` output stack diagrams (default OFF).

When this flag is ON and stack reporting is enabled, debugging information presenting a visual aid of how stack memory was allocated within a function will be displayed. For example, for the following function:

```
void f(int x) {
    double w;
    {
        int a;
        int b;
        int c;
    }
    float f;
    {
        double r;
        double y;
```

```
        }
    }
```

the +fsd option will produce:

```
### Auto Usage Stack Diagram for 'f' ###
x [+4] -> 4, 4
{
    w [+8] -> 12, 12
    {
        a [+4] -> 16, 16
        b [+4] -> 20, 20
        c [+4] -> 24, 24
    } [-12] -> 12, 24
    f [+4] -> 16, 24
    {
        r [+8] -> 24, 24
        y [+8] -> 32, 32
    } [-16] -> 16, 32
} [-12] -> 4, 32
###
```

where each new allocation (or complete block) is displayed in the format:

```
name [+added_size] -> current_usage, running_maximum_usage
```

`fse` use smallest underlying type for enums (default OFF).

In C, the underlying type of enumeration is implementation-defined but must be large enough to represent all the values in the enumeration as long as all of those values can be represented in an `int`. If all the values can be represented in a smaller `signed` or `unsigned type`, that smaller type may be used. In C++, the enumeration's underlying type is the first of the following types that can represent all of the provided values: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`. This flag indicates that the smallest type that can represent all of the values specified in the enumeration should be used as the underlying type.

`fsf` display function names for semantics during calls (default OFF).

When this flag is ON, every encountered function call will be accompanied by info 879, which provides all of the ways that the call may be specified inside of `-sem`, `-printf`, and `-scanf` options. This is useful when attempting to provide semantics for specific function overloads or instantiations where the precise syntax may not be obvious.

`fsi` search `#include` stack (default OFF).

When this flag is ON, in addition to searching the current working directory and the paths specified via `-i` options, header files specified with quoted syntax (e.g. `#include "a.h"`, not `#include <a.h>`), are searched for in each of the directories of the current include stack, starting from the top.

`fsl` single line comments (default OFF).

This flag controls whether C++-style comments are available in C89 mode.

`fsn` treat strings as names (default ON).

When the flag is OFF, the `esym()` option can be used only to suppress messages parameterized by '*Symbol*'. With this flag ON, `esym()` can also be employed to suppress (or enable) messages in the same way that `-estring()` can, e.g. messages parameterized by anything other than '*Type*' (for which `-etype()` must be used). For example:

```
//lint +fsn
//lint -esym(650, <)
int f(unsigned char c)
{
    if (c < 1000) return 1;  // Potential 650 ("constant out of
                             // range for operator 'String'")
    else return 0;
}
```

The `-esym()` second argument means that the `650` will not be issued when the operation (represented by the '*String*' parameter) is `"<"`.

`fso` return semantics override deduced return values (default OFF).

In previous versions, PC-lint would preserve the information in a user-defined return semantic even when more precise information was known about this value. The default behavior is to retain the more specific information. When the `fso` flag is ON, the PC-lint behavior is used. This applies only to functions with an implementation visible to PC-lint Plus.

`fsp` specific calls (default ON).

If this flag is ON (it is by default), Specific function call walking is supported. See Section 8.8 Interfunction Value Tracking. By turning this flag OFF (using the option `-fsp`), processing can be speeded up.

`fsv` track static variables (default ON).

Controls value tracking of static variables. When this flag is ON, all static variables will be tracked. If the flag is OFF, static variables will not be tracked between modules. Decrementing the flag when it is already OFF will disable tracking of static variables even within a single module.

`fta` enable typographical ambiguity checks (default ON).

When this flag is ON, MISRA C 2012 typographical ambiguity calculations are performed. If you are not interested in these checks and do not want to incur the associated overhead of this feature, you can turn this flag off. See also message `9046`.

`ftg` permit trigraphs (default ON).

If this flag in ON (it is ON by default) standard C/C++ trigraphs are permitted and message `854` is issued when they are converted. For example `??(` is a trigraph that denotes the left square bracket (`[`).

If this flag is OFF, trigraphs will not be converted and trigraph sequences will instead result in the issuance of warning `584`.

`fum` user declared move deletes only corresponding copy (default OFF).

In Standard C++, a user-declared move operation causes both the copy constructor and copy assignment function to be deleted. When this flag is set, only the corresponding operation will be deleted, e.g a user-declared move constructor will not result in the copy assignment function being deleted. This option is provided to support MSVC behavior.

`fun` issue additional stack usage notes (default OFF).

If this flag is ON, note `2901` (which must be separately enabled) will be issued with information about each function. This outputs the same information as note `974`, but is not restricted to the worst-case function.

`fur` allow unions to contain reference members (default OFF).

C++ forbids unions to contain reference members and by default PC-lint Plus does not allow this either. If this flag is ON, reference members will be accepted inside of unions.

`fvd` interactive value tracking debugger (default OFF).

This flag enables the value tracking debugger. See section 8.7 Debugger

`fwu` `wchar_t` is unsigned (default OFF).

This flag is used to specify the signedness of a built-in `wchar_t` type. If this flag is set ON the built-in type is of the unsigned variety, otherwise of the signed variety.

`fzd` enable sized deallocations (default OFF).
`fzl` `sizeof` is `long` (default OFF).

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

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

```
+fzl ++fzl
```

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

`fzu` `sizeof` is unsigned (default ON).

If this flag is ON, `sizeof()` is assumed to return an unsigned quantity (`unsigned long` if `fzl` is also ON). This flag is automatically adjusted upon encountering a `size_t` type.

4.11 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 Plus, the key to coping with these differences is the selection and/or modification of one or two compiler-specific files provided in the distribution.

```
co-xxx.lnt
```

This is used to specify lint options for a particular compiler. You may safely modify these files. For example `co-xxx.lnt` effectively contains the option

```
-esym( 534, fclose )
```

This inhibits the Warning message (534) that you would otherwise get if you called `fclose` and did not check the return value for errors. But if your programming policy is to always check the return value of this function you could remove this option. Alternatively you can negate its effect with a `+esym(534, fclose)` (after the first option was issued).

Compiler specific options files are provided with PC-lint Plus and the most current files are available from our website.

If your compiler is not supported with a `co-*.lnt` file, you may want to modify `co.lnt`, which is the generic compiler options file.

4.11.1 Customization Facilities

The following are useful for supporting a number of features in a variety of compilers. With some exceptions, they are used mostly to get PC-lint Plus to ignore some nonstandard constructs accepted by some compilers.

`@` Compilers for embedded systems frequently use the `@` notation to specify the location of a variable.

> We have for this reason support for the `@` feature, which consists of ignoring expressions to its right. When we see a '`@`' we then give a warning (430), which you may suppress with a `-e430`. For example:
>
> ```
> int *p @ location + 1;
> ```
>
> Although warning 430 is issued, `p` is regarded as a validly initialized pointer to `int`.

`_bit` is a type that is one bit wide.

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

`_gobble` is a reserved word that needs to be activated via `+rw(_gobble)`.

> It causes the next token to be gobbled; i.e., it and the next token are ignored. This is intended to be used with the `-d` option. See `co-kcarm.lnt` for examples.

`_ignore_init` This keyword when activated causes the initializer of a data declaration or the body of a function to be ignored.

> Cross compilers for embedded systems frequently have declarations that associate addresses with variables. For example, they may have the following declarations

```
Port pa = 0xFFFF0001;
Port pb = 0xFFFF0002;
```

etc. The type `Port` is, of course, non-standard. The programmer may decide to define `Port`, for the purpose of linting, to be an `unsigned char` by using the following option:

```
-d"Port=unsigned char"
```

(The quotes are necessary to get a blank to be accepted as part of the definition.) However, PC-lint Plus gives a warning when it sees a small data item being initialized with such large values. The solution is to use the built-in reserved word `_ignore_init`. It must be activated using the `+rw` option. Then it is normally used by embedding it within a `-d` option. For the above example, the appropriate options would be:

```
+rw(_ignore_init)
 -d"Port=_ignore_init unsigned char"
```

The keyword `_ignore_init` is treated syntactically as a storage class (though for maximum flexibility it does not have to be the ONLY storage class). Its effect is to cause PC-lint Plus to ignore, as its name suggests, any initializer of any declaration in which it is embedded.

Some compilers allow wrapping a C/C++ function prototype around assembly language in a fashion similar to the following:

```
__asm int a(int n, int m)
{ xeo 3, (n)r ; ... }
```

Note there is a special keyword that introduces such a function. This keyword may vary across compilers. To get PC-lint Plus to ignore the function body, equate this keyword with `_ignore_init`. E.g.

```
+rw(_ignore_init)
-d__asm = _ignore_init
```

`_to_brackets` is a reserved word that will cause it and the immediately following bracketed, parenthesized or braced expression, if any, to be ignored.

It needs to be activated with `+rw(_to_brackets)`. It is usually accompanied with a `-d` option. (For example, see `co-iar.lnt` on the distribution media). For example, the option:

```
-dinterrupt=_to_brackets
+rw(_to_brackets)
```

will cause each of the following to be ignored.

```
interrupt(3)
interrupt[5,5]
interrupt{x,x}
```

`_to_eol` When `_to_eol` is encountered in a program (or more likely some identifier defined to be `_to_eol`), the identifier and all remaining information on the line is skipped.

That is, information is ignored to the End Of Line. E.g., suppose the following nonstandard construct is valid for some compiler:

```
int f( int n ) registers readonly ( 3, 4 )
{
return n;
}
```

Then the user may use the following options so that the rest of the line following the first ')' is ignored:

```
-dregisters=_to_eol
+rw(_to_eol)
```

`_to_semi` is a super gobbler that will cause PC-lint Plus to ignore this and every token up to and including a semi-colon.

It needs to be enabled with `+rw(_to_semi)` and needs to be equated using `-d`. For example, if keyword `_pragma` begins a semicolon-terminated clause that you want PC-lint Plus to ignore, you would need two options:

```
-d_pragma=_to_semi
+rw(_to_semi)
```

`_up_to_brackets`
is a potential reserved word that will cause it and all tokens up to and including the next bracketed (or braced parenthesized) expression to be ignored. For example:

```
//lint +rw(_up_to_brackets)    activate reserved word
//lint -dasm=_up_to_brackets   asm is now an _up_to_brackets
asm ( "abc" : "def" );     // "asm" ... ')' is ignored
asm volatile ( "asm" );    // "asm" ... ')' is ignored
```

In the above we almost could have defined `asm` to be a `_to_brackets`. The problem is that we also needed to ignore the volatile following `asm` and so we required the use of `_up_to_brackets`.

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

Since it is not part of standard C or C++ the reserved word must be activated with the option:

```
+rw( __typeof__ )
```

`__typeof__` can be useful in macros where the exact type of an argument is not known.

For example:

```
#define SWAP(a,b) { __typeof__(a) x = a; a = b; b = x; }
```

will serve to swap the values of `a` and `b`. Some compilers not only support the `__typeof__` facility but they write their headers in terms of it. For example,

```
typedef __typeof__(sizeof(0)) size_t;
```

assures that `size_t` will not be out of synch with the built-in type.

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

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

`-d`*name*`{`*definition*`}` has the advantage that blanks may be embedded in the definition. Now it is true that you could use `-d"`*name*`=`*definition*`"` and so enclose blanks in that fashion but there are certain conditions, especially compiler generated macro definitions where the use of quotation marks are not suitable.

`-dname()=`*Replacement*
`-dname(`*identifier-list*`)=`*Replacement*

To induce PC-lint Plus to ignore or reinterpret a function-like sequence it is only necessary to `#define` a suitable function-like macro. However, this would require modifying source code (or use of the `-header` option) and is hence not as convenient as using this option. For example, if your compiler supports

```
char_varyingn
```

as a type and you want to get PC-lint Plus to interpret this as `char*` you can use

```
-dchar_varying()=char*
```

As another example:

```
//lint -dalpha(x,y)=((x+y)/x)
int n=alpha(2,10);
```

will initialize `n` to 6. The above `-dalpha`... option is equivalent to:

```
#define alpha(x,y) ((x+y)/x)
```

In the no parameter case, the functional expression can have any number of arguments. For example; in the following code both `asm()` expressions are ignored even though they have a different number of arguments.

```
//lint -dasm()=

void f()
{ asm("Move a,2", "Add a,b");
  asm("Jmp.x");
}
```

As with the normal (non-functional) version of the `-d` option the `+d` variant of the option sets up a macro that cannot be redefined.

4.11.2 Identifier Characters

Additional identifier characters can be established. See `-$` and `-ident()` in Section 4.4.3 Tokenizing.

4.11.3 Preprocessor Statements

See Section 12.4 Non-Standard Preprocessing for special non-standard preprocessor statements. Also see the `+ppw` option.

4.11.4 In-line assembly code

Compiler writers have shown no dearth of creativity in their invention of new syntax to support assembly language.

In the PC world, the most frequently used convention is (to simplify slightly):

```
asm { assembly-code }
```

or

```
asm assembly-code <new-line>
```

where `asm` is sometimes replaced with either `_asm` or `__asm`. This convention is supported automatically by enabling the asm keyword, using `+rw(asm)` (or `+rw(_asm)` or `+rw(__asm)` as the case may be).

But other conventions exist as well. One manufacturer uses

```
#asm
    assembly-code
#
```

For this sequence, it is necessary to enable `asm` as a pre-processor word using `+ppw(asm)`

If your compiler uses a different preprocessor word, you may use the option `+ppw_asgn`.

Another sequence is:

```
#asm
    assembly-code
#endasm
```

For this `+ppw(asm,endasm)` is needed.

Yet another convention is:

```
#pragma asm
    assembly-code
#pragma endasm
```

For this you need to define the two pragmas with

```
+pragma(asm, off)
+pragma(endasm, on)
```

4.11.5 Pragmas

4.11.6 Built-in pragmas

A number of compilers support the `push_macro` and the `pop_macro` pragmas.

`push_macro(` *`name-in-quotes`* `)`, where the *name-in-quotes* specifies a macro, is a pragma that will save the definition of the macro onto a stack. This will allow the macro to be redefined or undefined over a sub-portion of a module. Presumably this will be followed by a `pop_macro(` *`name-in-quotes`* `)` pragma that will restore the original macro. Thus:

```
#define N 100
#pragma push_macro( "N" )
#define N 1000
int x[N];
#pragma pop_macro( "N" )
int y[N];
```

declares `x` to be an array of 1000 integers whereas `y` becomes an array of 100 integers.

Note that you have, in effect, `k` different stacks where `k` is the number of different names provided as arguments to these pragmas.

4.11.7 User pragmas

`+pragma(` *`identifier,Action`* `)` associates *Action* with *identifier* for `#pragma`
`-pragma(` *`identifier`* `)` disables pragma *identifier*

The `+pragma(` *`identifier, Action`* `)` option can be used to specify an *identifier* that will be used to trigger an *Action* when the *identifier* appears as the first identifier of a `#pragma` statement. *Action* must be one of

```
on
off
once
message
```

```
ppw
macro
fmacro
options
include_alias
```

Please note that the purpose of the `+pragma` option is compatibility with your compiler. If your goal is to conditionally compile depending on the presence of PC-lint Plus, use the `_lint` preprocessor variable.

on and **off** – An `off` action will turn processing off. An `on` option will reset processing. For example, assume that the following coding sequence appears in a user program:

```
#pragma ASM
    movereg 3,8(4)
#pragma ENDASM
```

Lint will normally ignore the `#pragma` statements but it will not ignore the assembly language between the `#pragma` statements, which might lead to a flurry of messages. To resolve the problem, add the pair of options:

```
+pragma( ASM, off )
+pragma( ENDASM, on )
```

This will turn off lint processing when the `ASM` is seen and turn it back on when the `ENDASM` is seen.

Please do not get this backwards. See Section 4.11.4 In-line assembly code.

once: The option `+pragma(`*identifier*`,once)` allows the programmer to establish an arbitrary *identifier* (usually the identifier `once`) as an indicator that the header is to be included just once. To mimic the Microsoft C++ compiler use the option: `+pragma(once,once)`. Then, a subsequent appearance of the pragma:

```
#pragma once
```

within a header will cause that header to be included just once. Subsequent attempts to include the header within the same module will be ignored.

message: The option `+pragma(`*identifier*`, message)` allows the programmer to establish an arbitrary *identifier* (usually the identifier `message`) as a pragma that will produce a message to standard out. For example:

```
+pragma(message,message)
```

will cause the pragma:

```
#pragma message "hello from file" __FILE__
```

to write to standard out a greeting identifying the file within which the pragma is contained. As this example shows, macros are expanded as encountered in the message line. Also, messages fall under control of the conditional compilation statements, `#if`, etc. Following the Microsoft compiler, if the first token is a left parenthesis then only the parenthetical expression will be output. Other information on the line is not output.

ppw: The option `+pragma(` *indentifier*`, ppw )` will endow identifier with the `ppw` pragma action, which means reprocess the line as a preprocessor statement but with the word `"pragma"` ignored. Thus:

```
//lint +pragma( include, ppw )
#pragma include "abc.h"
```

will give the pragma keyword `"include"` the pragma action `ppw`, which will cause the next line to operate just like a standard `#include` preprocessor line.

macro, **fmacro** and **options**: Pragmas provide an avenue for the programmer to communicate to a compiler that is not governed by the syntax of the language. In most cases PC-lint Plus can ignore this information. Occasionally, however, the programmer will want us to act on this information so that he, the programmer, is not inserting the information twice, once for his compiler and once for PC-lint Plus.

Obviously it is impossible to provide compatible pragma recognition for all compilers now and into the future. The best general method of handling this seems to be to convert the pragma into a macro. Through the macro definition process, the macro can then become whatever the programmer wants. In particular it can become a `/*lint` *options...*`*/` comment and thereby be converted into a PC-lint Plus option. Alternatively it can be converted into code.

There are three basic ways of converting a pragma into a macro; these are identified as pragma types `macro`, `fmacro` and `options`.

macro: If a pragma is identified as `macro` as in the option

```
+pragma( identifier , macro )
```

then when a pragma by that name is encountered, the name is prefixed with the string '`pragma_`' and all the information to the right of the *identifier* is enclosed in parentheses. For example, one compiler supports statements such as:

```
#pragma port x @ 0x100
```

This would identify `x` as a particular I/O port. Later in the program there may be assignments to or from `x`. If PC-lint Plus were to ignore the pragma, it would have to emit syntax errors on every use of `x`.

If the option `+pragma( port,macro )` is given, the above pragma will be converted into:

```
pragma_port( x @ 0x100 )
```

Presumably there is a macro definition that resembles:

```
#define pragma_port( s ) volatile unsigned s;
```

Such a definition can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

fmacro: The second form of macroizing a pragma is identified as `fmacro` (meaning function macro). This would be used in a `+pragma` option having the form:

```
+pragma( identifier , fmacro )
```

With the `fmacro` type, instances of the pragma are assumed to already be in functional notation. Like the macro type, the name of the pragma is prefixed with `pragma_` to avoid conflicts with other uses of the pragma name. Aside from this prefixing the pragma is taken as found in the pragma statement and employed as a macro. For example, consider a pragma called `warnings`, which appears as:

```
#pragma warnings(no)
```

or

```
#pragma warnings(yes)
```

Presumably the `no` form turns off warnings and the `yes` form turns them back on.

If the option `+pragma(warnings,fmacro)` is given, the first pragma will be converted into:

```
pragma_warnings(no)
```

and the second will be converted into:

```
pragma_warnings(yes)
```

The programmer will want to provide a set of macros that can convert these pragmas into the equivalent form for PC-lint Plus. This could be done as follows:

```
#define pragma_warnings(x) pragma_warnings_##x
#define pragma_warnings_no /*lint -save -w1 */
#define pragma_warnings_yes /*lint -restore */
```

These macro definitions can be placed in a header file that only PC-lint Plus will see by utilizing the `-header` option.

options: The third form of pragma that can be macroized is identified as '`options`' using an option of the form:

```
+pragma( identifier , options )
```

When a pragma having the name *identifier* is used, it is presumably followed by a blank-separated sequence of options having the form *name*=*value*. An example is:

```
#pragma OPTIONS tab=4 length=80
```

In this form, each individual option becomes a potential macro invocation. The name of this macro is formed by concatenating `pragma_`, the *identifier*, which in this case is `OPTIONS`, followed by underscore and the name of the suboption. Thus for the suboption `tab` the name of the macro is `pragma_OPTIONS_tab`. As an example, suppose the option `+pragma( OPTIONS, options )` is given and the following macro defined.

```
#define pragma_OPTIONS_tab(x) /*lint -t##x */
```

Then the above pragma will result in just the single macro invocation:

```
pragma_OPTIONS_tab(x)
```

This will invoke the tab option of PC-lint Plus. The '`length=80`' option is ignored. To attach meaning to the length option it would only be necessary to define a macro whose name would be `pragma_OPTIONS_length`.

include_alias: The option `+pragma(`*identifier*`, include_alias)` allows the programer to indicate an arbitrary identifier as mimicking the Microsoft C/C++ compiler pragma `include_alias`. For example:

```
+pragma(alias_include, include_alias)
```

will cause the pragmas:

```
#pragma alias_include( "a.h", "123.h" )
```

to inform Lint, when a `#include` directive appears for `"a.h"`, to search for `"123.h"` instead.

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

```
-pragma(message)
```

will remove the `message` pragma.

5 Libraries

Please note: This chapter is not about how to include header files that may be in some directory other than the current directory. For that information see the `-i` option (Section 4.4.2 Preprocessor) or Section 12.2.1 INCLUDE Environment Variable. This chapter explains how information in header files (and possibly modules) is interpreted.

Examples of libraries are compiler libraries such as the standard I/O library, and third-party libraries such as windowing libraries, and database libraries. Also, an individual programmer may choose to organize a part of his own code into one or more libraries if it is to be used in more than one application. The important features of libraries, in so far as linting is concerned, are:

(*a*) The source code is usually not available for linting.

(*b*) The library is used by programs other than the one you are linting.

Therefore, to produce a full and complete analysis it is essential to know which headers represent libraries. It is also possible for modules to be available for linting but, because they are created beyond the control of the immediate programmer, they too can benefit from the designation 'library'.

5.1 Library Header Files

A library header file is a header file that describes (in whole or in part) the interface to a library.

The most familiar example of a library header file is `stdio.h`. Consider the file `hello.c`:

```
#include <stdio.h>

int main(void) {
    printf( "hello world\n" );
}
```

Without the header file, PC-lint Plus would complain that `printf` was neither declared (Informational `718`) nor defined (Warning `526`). (The distinction between a declaration and a definition is extremely important in C/C++. A definition for a function, for example, uses curly braces and there can be only one of them for any given function. Conversely, a declaration for a function ends with a semi-colon, is simply descriptive, and there can be more than one).

If `hello.c` were a C++ program an even stronger message would be issued, but we will assume a straight C program.

With the inclusion of `stdio.h` (assuming `stdio.h` contains a declaration for `printf`), PC-lint Plus will not issue message `718`. Moreover, if `stdio.h` is recognized as a library header file, (it is by default because it was specified with angle brackets), PC-lint Plus will understand that source code for `printf` is not necessarily available and will not issue warning `526` either. Note: Other messages associated with library headers are not suppressed automatically. But you may use `-wlib` or any of the `-elib`... options for this purpose. See Section 4.3.1 Error Inhibition.

A header file can become a library header file if:

(*a*) It falls within one of the four broad categories of the option `+libclass`, viz. `all`, `ansi`, `angle` and `foreign` (described below), and is not excluded by either the `-libdir` or the `-libh` option.

(*b*) OR, for finer control, it comes from a directory specified with `+libdir` and is not specifically excluded with `-libh`.

(*c*) OR, for the finest control, it is specifically included by name via `+libh`.

(*d*) OR, is included within a library header file.

You may determine whether header files are library header files by using some variation of the `-vf` verbosity option. For each included library header you will receive a message similar to:

```
Including file c:\compiler\stdio.h (library)
```

The tag: '(`library`)' indicates a library header file. Other header files will not have that tag.

What follows is a more complete description of the three options used to specify if or when a header file is a library header file.

`+libclass(` *`identifier`* `[,` *`identifier`*`] ...)` specifies the set or sets of header files that are assumed to be library header files.

Each identifier can be one of:

`angle` All headers specified with angle brackets.

`foreign` All header files found in directories that are on the search list (`-i` or `INCLUDE` as appropriate).

Thus, if the `#include` contains a complete path name then the header file is not considered '`foreign`'. To endow such a file with the library header property use either the `+libh` option or angle brackets. For example, if you have

```
#include "\include\graph.h"
```

and you want this header to be regarded as a library header use angle brackets as in:

```
#include <\include\graph.h>
```

or use the option:

```
+libh(\include\graph.h)
```

Similar remarks can be made about

```
#include "include\graph.h"
```

If a search list (specified with `-i` option or `INCLUDE`) is used to locate this file it is considered `foreign`; otherwise it is not.

`ansi` The 'standard' ANSI/ISO C header files, viz.

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

`all` All header files are regarded as being library headers.

By default, `+libclass(angle,foreign)` is in effect. This option is not cumulative. Any `+libclass` option completely erases the effect of previous `+libclass` options. To specify no class use the option `+libclass()`.

`+libdir(`*directory* `[,` *directory*`] ... )` activates
`-libdir(`*directory* `[,` *directory*`]...)` deactivates

the directory (or directories) specified. The notion of *directory* here is identical to that in the -i option. If a *directory* is activated then all header files found within the directory will be regarded as library header files (unless specifically inhibited by the -libh option). It overrides the +libclass option for that particular directory. For example:

```
+libclass()
+libdir( c:\compiler )
+libh( os.h )
```

requests that no header files be regarded as library files except those coming from directory `c:\compiler` and the header `os.h` (see below). Also,

```
+libclass( foreign )
-libdir( headers )
```

requests that all headers coming from any foreign directory except the directory specified by `headers` should be regarded as library headers.

Wild card characters '*' and '?' are supported. Note: A file specified as

```
#include "c:\compiler\i.h"
```

is not regarded as being a library header even though `+libdir(c:\compiler)` was specified. Only files found in `c:\compiler` via a search list (-i or `INCLUDE`) are so regarded and only when the -i option matches the `libdir` parameter. For example,

```
#include "compiler\i.h"
```

will also not be considered as library even though the -ic: option is given, and the file is found by searching. The -i search directory (`c:`) is not matching the `libdir` directory (`c:\compiler`).

`+libh(` *file*`[,` *file*`] ... )` adds
`-libh(` *file*`[,` *file*`]...)` removes

files from the set that would otherwise be determined from the +libclass and -/+libdir options. For example:

```
+libclass( ansi, angle )
+libh( windows.h, graphics.h )
+libh( os.h ) -libh( float.h )
```

requests that the header files described as `ansi` or `angle` (except for `float.h`) and the individual header files: `windows.h`, `graphics.h` and `os.h` (even if not specified with angle brackets) will be taken to be library header files.

Wild card characters '*' and '?' are supported.
For `libh` to have an effect, its argument must match the string between quotes or angle brackets in the `#include` line. Thus in the case of:

```
#include <../lib/graphics.h>
```

you must have `+libh(../lib/graphics.h)`.

Note that the `libh` option is accumulative whereas the +libclass option overrides any previous +libclass option including the default.

When a `#include` statement is encountered, the name that follows the `#include` is defined to be the *header-name* (even if the name is a compound name containing directories). When an attempt is made to open the file, a list of directories is consulted, which are all those specified by -i options and the `INCLUDE` environment variable. The directory that is used to successfully open the file is defined to be the *header-directory*.

The options +libdir(...) and -libdir(...) are applied to the **header-directory** and the options +libh(...) and -libh(...) are applied to the *header-name* (as defined in the previous paragraph). For example, given the following:

```
#include "graphics\shapes.h"
```

Suppose that the following option had been given:

```
-iC:\
```

and suppose further that a file "`C:\graphics\shapes.h`" exists. Then the *header-name* would be "`graphics\shapes.h`" and the *header-directory* would be "`C:\`". Any one of the following options could be used to designate the file as a library file.

```
+libh( graphics\* )
+libh( *shapes.h )
+libdir( C:* )
+libdir( C:\ )
```

5.2 Library Modules

You may designate that a module is a library module using the option:

```
+libm( module-name )
```

You would normally use just the '+' form of the option. But you may use `-libm` to undo the effects of a `+libm` option with some arguments.

This option has the effect of designating the entire module and all of the header files that it includes, as "library". That is, messages will be inhibited via -wlib or -elib... options. Unused globals defined within such a module will draw no complaints, etc.

As an example, suppose you have an application `alpha.c`, and that this code requires the services of a module `beta.c`, which is generated by a separate program. Typically the interface to `beta.c` will be described by a header file `beta.h` and a typical linting can be specified by:

```
lint +libh( beta.h ) alpha.c
```

But another possibility is to include `beta.c` in the lint. This would have the advantage of facilitating inter module value tracking. The typical command to do this would be:

```
lint +libh( beta.h ) +libm( beta.c ) alpha.c beta.c
```

Note that the option `libm` takes a pattern that may include wild-cards. Let us suppose that our generator will generate not just `beta.c` but a sequence of three modules

```
beta1.c   beta2.c   beta3.c
```

Then they can all be designated as library with the single option

```
+libm( beta*.c )
```

5.3 Assembly Language Modules

In this section we deal with the case of assembly-language modules. For in-line assembly code see Section 4.11.4 In-line assembly code.

If one or more modules of your application are written in assembly language or, equivalently, in some language other than C or C++ (a common phrase is "mixed language"), you must arrange so that the missing code does not cause PC-lint Plus to give spurious messages. The most common way of proceeding is to create a header file describing the assembly language portion of your application. This header file, say `asm.h`, will have property (a) of library header files in that the objects declared therein will not be defined in files seen by PC-lint Plus. Hence we make it a library header file with the option:

```
+libh(asm.h)
```

Finally, the assembly language portion of your application may be the only portion of your application that is referencing, initializing or accessing some variable or function. A spurious "not referenced" or "not accessed" message would be given. The easiest thing to do is to explicitly suppress the message(s). For example, if the assembly language portion is the only portion accessing variable `alpha` and you are getting message `552`, then place option `-esym(552,alpha)` among your lint options. If you are using our suggested setup, as described in Section 13.2 Recommended Setup, then `std.lnt` will now have the contents:

```
c.lnt
options.lnt
+libh(asm.h)
-esym(552,alpha) //accessed in assembly language
```

You might be tempted to place these options in lint comments within `asm.h`. Unfortunately, the `libh` option will be set too late to establish `asm.h` as a library header.

You might yet say that "My assembly language routines are sometimes opted out and sometimes opted in, and this is under control of a global preprocessor variable `USEASM`. When opted out, C/C++ equivalent routines are activated. How can I cope with this varying situation?"

This actually makes the situation easier. Just make sure that when you are linting, `USEASM` is opted out. You might use:

```
#ifdef _lint
#undef USEASM
#endif
```

or some equivalent sequence. In this way, lint will know the intent of the assembly code from the equivalent C/C++ code. The previously suggested options of `+libh` and `-esym` are then not necessary.

6 Precompiled Headers

6.1 Introduction to precompiled headers

Most readers of this information will already be familiar with the notion of a precompiled header. With traditional precompiled headers, a single header is designated as one to be precompiled. In PC-lint, such a file was one that was expected to be `#include`'d in the source module. In PC-lint Plus, an existing precompiled header is processed as a prefix header and is loaded before each module that follows the pch option designating the PCH header, regardless of whether that module explicitly `#includes` the header or not. PC-lint Plus's precompiled headers act as a sort of "on disk caching" of a previously compiled header file. Information is loaded into memory as needed, reducing processing time.

Two types of precompiled headers exist; those precompiled for C modules have the extension "`lcph`' while those precompiled for C++ modules have the extension "`lpph`". For example, if a header's name is "`a.h`" and it is being precompiled for a C module, the resultant file will be "`a.lcph`". If a specific header is marked for precompilation and a precompiled version does not already exist, PC-lint Plus will precompile that header at the start of examining a module, saving it to a file with the relevant extension. PC-lint Plus will create a C or C++ version of the precompiled header only if necessary to process the next module. Otherwise, the file will be opened and read in as needed.

Note: if you `#include` the file from which a precompiled header is created, you are advised to follow what is typically considered good programming practice and make sure that header contains a standard include guard.

To designate that a header is to be precompiled use the option:

-pch(*header-name*)

The *header-name* should be the name of a file that can be found via the traditional include process, such as by examining `-i` options.

If you want to use a precompiled header for some modules and not for others, you can disable the use of a precompiled header with the option:

```
-pch()
```

For example, to use header "`a.h`" as a precompiled header for the first three modules of your project and not the fourth, the arguments passed to PC-lint Plus should look something like this:

```
-pch(a.h)
module1.c      //  creates ‘a.lcph‘ if needed
module2.cpp    //  creates ‘a.lpph‘ if needed
module3.c      //  ‘a.lcph‘ already exists; load as needed
-pch()
module4.c      //  neither create nor use any precompiled header
```

You can also specify multiple precompiled headers per project, though only one per module. You do so by passing another `-pch(`*header-name*`)` styled option to PC-lint Plus after the previous module name and before the next. If we alter our example above to use a precompiled header for "`b.h`" in the fourth module of the project, the argument list would look something like this:

```
-pch(a.h)
module1.c      //  creates ‘a.lcph‘ if needed
module2.cpp    //  creates ‘a.lpph‘ if needed
module3.c      //  ‘a.lcph‘ already exists; load as needed
-pch(b.h)
module4.c      //  creates ‘b.lcph‘ if needed
```

6.2 Designating the precompiled header

To designate that a header is to be precompiled use the option:

```
-pch( header-name )
```

The *header-name* should be that name used between angle brackets or between quotes on the `#include` line. In particular, if the name on the `#include` line is not a full path name do not use a full path name in the option.

Normally a precompiled header is the first header encountered in each of the modules that include it. Occasionally it is not, because the `-header()` option forcefully (if silently) includes a header just prior to the start of each module. Also, it just might be desirable to include a header prior to the one declared to be the precompiled header. So earlier headers are permitted. But if a precompiled header does follow an include sequence, it must follow that same include sequence in every module in which it is included. Otherwise a diagnostic will be issued.

6.3 Monitoring precompiled headers

The sequence of events that takes place when a precompiled header is included can be monitored by using a variant of the verbosity option that contains or implies the letter '`f`'. Given the option sequence:

```
-pch(x.h) -vf
```

we would expect to see, at the first time `x.h` is included, the verbosity line:

```
Including file x.h (bypass)
```

As indicated above, `x.h` becomes, of necessity, a file to be bypassed in subsequent modules. After fully processing `x.h` and all of its includes we will see the line:

```
Outputting to file x.lph
```

The extension "`lph`" stands for "lint precompiled header". The name of the file containing the precompiled output is formed by appending this extension onto the root of the file named in the `pch` option.

In subsequent modules you will see the verbosity line:

```
Bypassing x.h
```

in place of a line that would normally show an include of this header.

If the program were to be linted subsequently with the same options, then instead of seeing a verbosity line indicating that `x.h` were included and `x.lph` were written we would see:

```
Absorbing file x.lph
```

reflective of the fact that `x.lph` contains binary information representative of the information in `x.h`.

6.4 The use of make files

The `.lph` file is not automatically regenerated when the original header (or any of its sub headers) is modified. If it is important that it must be done automatically then you will need a `make` facility or its equivalent. An entry in the `make` file could be as simple as:

```
x.lph:  x.h ...
        del x.lph
```

In words, an `x.lph` is composed of `x.h` plus any of its included header files and is 'manufactured' by a deletion of `x.lph`. If this confuses the `make` facility then you might try something like:

```
request.lph:  x.h ...
        del x.lph
        touch request.lph
```

Here you need to create a file called "`request.lph`" whose content is the minimal necessary for `make` to consider it a file. Whenever any of a collection of headers is modified, `x.lph` is deleted and the date of the `request.lph` is updated.

7 Strong Types

Strong type checking is gold
Normal type checking is silver
But casting is brass

7.1 Rationale

The strong type system allows you to imbue typedefs with flexible type-checking properties and can perform dimensional analysis. For example, consider the law of universal gravitation:

$$F = G\frac{m_1 m_2}{r^2}$$

The following code attempts to implement this, but contains a mistake:

```
typedef double Meter, Second, Velocity, Acceleration;
typedef double Kilogram, Newton;
typedef double Area, Volume;
typedef double GravitationalConstant;

const GravitationalConstant G = 6.67e-11;

Newton attraction(Kilogram mass1, Kilogram mass2, Meter distance) {
    return (mass1 * mass2) / (distance * distance);
}
```

A compiler is not interested in (and has no obligation to warn you about) the dimensional mismatch here caused by forgetting to multiply by `G`. Running this example through lint with the appropriate strong type options produces the following messages:

```
strong type mismatch:  assigning '(Kilogram*Kilogram)/(Meter*Meter)' to 'Newton'
did you mean to multiply by a factor of type 'GravitationalConstant'?
```

If you are curious what options were used to get these units into the strong type system, see Full Source for the Gravitation Example.

7.2 Creating Strong Types with -strong

The primary option used to interact with the strong type system is

`-strong(`*flags*`[,`*name* `...])`

This option identifies each *name* as a strong type with properties specified by *flags*. Presumably there is a later `typedef` defining any such *name* to be a type. If no *name* is provided, the specified *flags* will be taken as the default for types without explicit `-strong` options. Flags are uppercase letters that indicate some aspect of a type's behavior, and they can be modified by following them with softeners. Softeners are represented using lowercase letters and must immediately follow the flag they are modifying.

`A` Check strong types on Assignment. Issue a warning upon assignment (where assignment refers to using the assignment operator, returning a value, passing an argument or initializing a variable). `A` may be followed by one or more softening modifiers:

- `i` ignore Initialization
- `r` ignore Return statements
- `p` ignore Passing arguments
- `a` ignore the Assignment operator
- `c` ignore assignment of Constants (literals)
- `z` ignore assignment of integer constant expressions equal to Zero (non-strong casts are ignored)

`X` Check strong types on eXtraction. This flag issues warnings in the same contexts as the `A` flag, but checks on behalf of the value being assigned. The softeners for `A` cannot be used with `X`.

`J` Check strong types when Joining two operands of a binary operator. `J` may be followed by one or more of the following modifiers:

- `e` ignore Equality operators, (`==` and `!=`), and the conditional operator, (`?:`)
- `r` ignore Relational operators, (`>`, `>=`, `<`, and `<=`)
- `m` ignore Multiplicative operators, (`*`, `/`, and `%`)
- `d` indicates that this strong type is a Dimension (see 7.4.1 Dimensional Types)
- `n` indicates that this strong type is dimensionally Neutral (see 7.4.2 Dimensionally Neutral Types)
- `a` indicates that this strong type is Antidimensional (see 7.4.3 Antidimensional Types)
- `o` ignore Other (non-multiplicative) binary operators, (`+`, `-`, `|`, `&`, and `^`)
- `c` ignore combining with constants
- `z` ignore combining with Zero, as in `Az` above

`B` Designate a major boolean type. Only one boolean type may exist whether it comes from the `B` or `b` flag. The result of all boolean operators will be a value compatible with this type. Contexts that expect a boolean value will require their operands to be of the major boolean type.

`b` Designate a minor boolean type. Only one boolean type may exist whether it comes from the `B` or `b` flag. The result of all boolean operators will be a value compatible with this type. This flag places no requirement on the values used in contexts that expect a boolean, in contrast to `B`.

`l` Designate a type as inherently compatible with library functions. This includes assignment from library function return values and as library function arguments.

`f` Indicates bit-fields of length one are not automatically boolean (by default they are). This is a modifier that can only accompany one of the boolean flags (either `B` or `b` above). .

7.3 Strong Types for Array Indices

Description

```
-index( flags, ixtype, sitype [, sitype ...] )
```

This option is supplementary to and can be used in conjunction with the `-strong` option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype* (or *sitype*'s if more than one is provided). Please note: both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a `typedef` declaration. *flags* can be

- `c` allow Constants as well as *ixtype*, to be used as indices.
- `d` allow array Dimensions to be specified without using an *ixtype*.

Examples of -index

For example:

```
//lint -strong( AzJcX, Count, Temperature )
//lint -index( d, Count, Temperature )
//       Only Count can index a Temperature

typedef float Temperature;
typedef int Count;
Temperature t[100];         // OK because of d flag
Temperature *pt = t;        // pointers are also checked

                            // ... within a function
Count i;

t[0] = t[1];                // Warnings, no c flag
for( i = 0; i < 100; i++ )
t[i] = 0.0;                 // OK, i is a Count
pt[1] = 2.0;                // Warning
i = pt - t;                 // OK, pt-t is a Count
```

In the above, `Temperature` is said to be *strongly indexed* and `Count` is said to be a *strong index*.

If the `d` flag were not provided, then the array dimension should be cast to the proper type as for example:

```
Temperature t[ (Count) 100 ];
```

However, this is a little cumbersome. It is better to define the array dimension in terms of a manifest constant, as in:

```
#define MAX_T (Count) 100
Temperature t[MAX_T];
```

This has the advantage that the same `MAX_T` can be used in the `for` statement to govern the range of the `for`.

Note that pointers to the Strongly Indexed type (such as `pt` above) are also checked when used in array notation. Indeed, whenever a value is added to a pointer that is pointing to a strongly indexed type, the value added is checked to make sure that it has the proper strong index.

Moreover, when strongly indexed pointers are subtracted, the resulting type is considered to be the common Strong Index. Thus, in the example,

```
i = pt - t;
```

no warning resulted.

It is common to have parallel arrays (arrays with identical dimensions but different types) processed with similar indices. The `-index` option is set up to conveniently support this. For example, if `Pressure` and `Voltage` were types of arrays similar to the array `t` of `Temperature` one might write:

```
//lint -index( , Count, Temperature, Pressure, Voltage )
...
Temperature t[MAX_T];
Pressure p[MAX_T];
Voltage v[MAX_T];
...
```

Multidimensional Arrays

The indices into multidimensional arrays can also be checked. Just make sure the intermediate type is an explicit `typedef` type. An example is `Row` in the code below:

```
/* Types to define and access a 25x80 Screen.
a Screen is 25 Row's
a Row is 80 Att_Char's */

/*lint -index( d, Row_Ix, Row )
-index( d, Col_Ix, Att_Char ) */

typedef unsigned short Att_Char;
typedef Att_Char Row[80];
typedef Row Screen[25];

typedef int Row_Ix; /* Row Index */
typedef int Col_Ix; /* Column Index */

#define BLANK (Att_Char) (0x700 + ' ')

Screen scr;
Row_Ix row;
Col_Ix col;

void main()
{
int i = 0;

scr[ row ][col ] = BLANK;        /* OK */
scr[ i ][ col ] = BLANK;         /* Warning */
scr[col][row] = BLANK;           /* Two Warnings */
}
```

In the above, we have defined a `Screen` to be an array of `Row`'s. Using an intermediate type does not change the configuration of the array in memory. Other than for type-checking, it is the same as if we had written:

```
typedef Att_Char Screen[25][80];
```

7.4 Dimensional Analysis

Unlike other binary operators that expect their operands to agree in strong type, multiplication and division often can and should handle different types in what is commonly referred to as dimensional analysis. But not all strong types are the same in this regard. The strong type system recognizes three different kinds of treatment with regard to multiplication and division.

7.4.1 Dimensional Types

A *dimension* is a strong type such that when two expressions are multiplied or divided and each type is a dimension, then the resulting type will also be a dimension whose name will be a compound string representing the product or quotient of the operands (reduced to lowest terms). The modulus operator `%` will have a resultant type equal to the type of the numerator.

For example:

```
//lint -strong( AJdX, Sec )
typedef double Sec;
Sec x, y;
...
x = x * y;      // warning: '(Sec*Sec)' is assigned to 'Sec'
y = 3.6 / x;    // warning: '1/Sec' is assigned to 'Sec'
```

Flags '`AJdX`' contain the Join phrase '`Jd`' designating that Sec is a dimension. Strictly speaking the '`d`' is not necessary because the normal default is to make any strong type dimensional. However, there is a flag option `-fdd` (turn off the Dimension by Default flag), which will reverse this default behavior, so it is probably wise to place the '`d`' in explicitly.

Dimensional types are treated in greater detail later.

7.4.2 Dimensionally Neutral Types

A *dimensionally neutral* type is a strong type such that when multiplied or divided by a dimension will act as a non-strong type.
For example:

```
//lint -strong( AJdX, Sec )
typedef double Sec;

//lint -strong( AJnX, Cycles )
typedef double Cycles;
Cycles n;
Sec t;
...
t = n * t;      // OK, Cycles are neutral
t = t / n;      // still OK.
n = n / t;      // warning: '1/Sec' assigned to 'Cycles'
```

The `n` softener of the `J` flag as in the `AJnX` sequence above designates that type `Cycles` is dimensionally neutral and will drop away when combined multiplicably with the dimension `Cycles` as shown in the first two assignments. However, `Cycles` acts as a strong type in every other regard. An illustration of this is the last line in this example, which produces a warning that the type '`1/Sec`' is being assigned to `Cycles`.

Thus, `Cycles` is playing the role that it traditionally plays in Physics and Engineering. It contains no physical units and when multiplied or divided by a dimension does not change the dimensionality of the result.

7.4.3 Antidimensional Types

An *antidimensional* type is a strong type that when multiplied or divided is expected to be combined with the same type, or one that is compatible through the usual strong type hierarchies. It functions in this regard much like addition and subtraction.

For example:

```
//lint -strong( AJaX, Integer )
typedef int Integer;
Integer k;
int n;
...
k = k * k; // OK
k = n * k; // warning: Integer joined with non-Integer
```

The sequence `Ja` in the above indicates that `Integer` is antidimensional.

7.4.4 Multiplication and Division of Dimensional Types

The strong type mechanism can support the traditional dimensional analysis exploited by physicists, chemists and engineers. When strong types are added, subtracted, compared or assigned, the strong types need merely match up with each other. However, multiplication and division can join arbitrary dimensional types and the result is often a new type. Consider forming the velocity from a distance and a time:

```
//lint -strong( AcJcX, Met, Sec, Velocity = Met/Sec )
typedef double Met, Sec, Velocity;
Velocity speed( Met d, Sec t )
    {
    Velocity v;
    v = d / t;          // ok
    v = 1 / t;          // warning
    v = (3.5/t) * d;    // ok
    v=(1/(t*t))*d*t;    // ok
    return v;// ok
    }
```

In this example, the 4th argument to the `-strong` option:

```
Velocity = Met/Sec
```

relates strong type `Velocity` to strong types `Met` and `Sec`. This particular suboption actually creates two strong types: `Velocity` and `Met/Sec` and relates the two types by making `Met/Sec` the parent type of `Velocity`. This relationship can be seen in the output obtained from the option `-vh` (or the compact form `-vh-`). As an example the results of the `-vh` option for the above example are:

```
-  Met
-  Sec
-  Met/Sec
   |
   + - Velocity
-  1/Sec
-  (Sec*Sec)
-  1/(Sec*Sec)
-  Met/(Sec*Sec)
```

The division of `Met` by `Sec` (within the option) can be produced in many equivalent ways. E.g.

```
Velocity = (1/Sec) * Met
Velocity = ((1/Sec) * (Met))
Velocity = (Met/(Sec*Sec)) * Sec
```

are all equivalent. All of these dimensional expressions are reduced to the canonical form `Met/Sec`, which was the form given in the original option. Note that parentheses can be used freely and in some cases must be used to obtain the correct results. E.g.

```
Acceleration = Met/Sec*Sec        // wrong
Acceleration = Met/(Sec*Sec)      // correct
```

We follow C syntactic rules where the operators bind left to right and the example labeled 'wrong' results, after cancellation, in just `Met`.

Briefly and for the record the canonical form produced is:

```
(F1*F2*...*Fn)/(G1*G2*...*Gm)
```

where each Fi and each Gi are simple single-identifier sorted strong types and where $n >= 0$ and $m >= 0$ but if n is less than 2 the upper parentheses are dropped out and if m is less that 2 the lower parentheses are dropped and if n is 0 the numerator is reduced to `1` and if m is 0 the entire denominator including the / is dropped.

Returning to our original example (the function `speed`), when the statement:

```
v = d/t;
```

is encountered and an attempt is made to evaluate `d/t` the dimensional nature of the types of the two arguments is noted and the names of these types is combined by the division operator to produce "`Met/Sec`". This uses essentially the same algorithms and canonicalization as the compound type analysis with a `-strong` option. The resulting type is assigned to `Velocity` without complaint because of the previously described parental relationship that exists between these two strong types.

In the next statement

```
v = (3.5/t) * d;
```

the division results in the creation of a new strong type (`1/Sec`), which when multiplied by `Met` will become `Met/Sec`. The created type will have properties `AJcdX` and the underlying type will be the type that a compiler would compute.

7.4.5 Dimensional Types and the % operator

Let's say you have a paper 400 lines long and the printing requires 60 lines/page. How many full pages will we require? The answer is

```
400 lines / (60 lines/page)
= 6 pages
```

How many lines are left over? The answer is

```
400 lines % (60 lines/page)
= 40 lines
```

Thus, unlike division, the `%` operator yields a dimension that equates to the dimension of the numerator (in this case, `lines`) while ignoring the dimension of the 2nd operand.

7.4.6 Conversions

A simple example in the use of Dimensional strong types is that of providing a fail-safe method of converting from one system of units to another. Such conversions can quite often be accomplished by a single numeric factor. Such conversion factors should have dimensions attached to prevent mistakes. E.g.

```
// Centimeters to/from Inches
//lint -strong( AJdX, In, Cm, CmPerIn = Cm/In )
typedef double In, Cm, CmPerIn;
CmPerIn cpi = (CmPerIn) 2.54;    // conversion factor
void demo( In in, Cm cm )
    {
    ...
```

```
    in = cm / cpi;              // convert cm to in
    ...
    cm = in * cpi;              // convert in to cm
    ...
    }
```

In this example we are defining a conversion factor, `cpi`, that will allow us to convert inches to centimeters (by multiplication) and convert centimeters to inches (via division). Without strong types, conversion factors can be misused. Do I multiply or divide? Using strong types you can be assured of getting it right.

Obviously not all conversions fall into the category of being described by a conversion factor. Conversions between Celsius and Fahrenheit, for example, require an expression and this typically means defining a pair of functions as in the following:

```
//lint -strong( AJdX, Fahr, Celsius )
typedef double Fahr, Celsius;
Celsius toCelsius( Fahr t )
    { return (t-(Fahr)32.) * (Celsius)5. / (Fahr)9.; }
Fahr toFahr( Celsius t )
    { return (Fahr)32. + t * (Fahr)9. / (Celsius)5.; }
```

The function call overhead is probably not significant, but if it is, you may declare the functions to be inline in C++. Some C systems support inline functions, but in any case, you can use macros.

Now let us suppose a confused programmer had written:

```
Fahrenheit f;
Celsius c;
...
f = toCelsius (c);    // Type Violations
```

Then there would be two strong type violations since passing `c` to a `Fahrenheit` variable is bad as is assigning a `Celsius` value to `f`.

7.4.7 Integers

Although the examples of dimensional analysis offered above refer to floating point quantities, the same principles apply to integer arithmetic. E.g.

```
#include <stdio.h>
#include <limits.h>
//lint -strong( AcJdX, Bytes, Bits )
//lint -strong( AcJdX, BitsPerByte = Bits / Bytes )
typedef size_t Bytes, Bits, BitsPerByte;
BitsPerByte bits_per_byte = CHAR_BIT;
Bytes size_int = sizeof(int);
Bits length_int = size_int * bits_per_byte;
```

In this example Bits is the length of an object in bits and `Bytes` is the length of an object in bytes. `bits_per_byte` becomes a conversion factor to translate from one unit to the other. The example shows the use of that conversion factor to compute the number of bits in an integer.

Let's say that you wanted to strengthen the integrity and robustness of a program by making sure that all shifts were by quantities that were typed `Bits`. For example you could define a function `shift_left` with the intention that this function have a monopoly on shifting `unsigned` types to the left. This could take the form:

```
inline unsigned shift_left( unsigned u, Bits b ) {
    return u << b;
```

```
}
```

A simple grep for "<<" can be used to ensure that no other shift lefts exist in your program. Note that the example deals only with `unsigned` but if there were other types that you wanted to shift left, such as `unsigned long`, you can use the C++ overload facility.

Using C you may also employ the `shift_left` function. However you may not have inline available and you may be concerned about speed. To obtain the required speed you can employ a macro as in:

```
#define Shift_Left(u,b) ((u) << (b))
```

But you will note that there is now no checking to ensure that the number of bits shifted are of the proper type. One approach is to use conditional compilation:

```
#ifdef _lint
#define Shift_Left(u,b) shift_left(u,b)
#else
#define Shift_Left(u,b) ((u) << (b))
#endif
```

This will work adequately in C. If the quantity being shifted is anything other than plain `unsigned`, you will need to duplicate this pattern for each type.

A probably better approach is to define a macro that can check the type, such as the macro `Compatible` defined below:

```
#ifdef _lint
  #define Compatible(e,type) (*(type*)__Compatible = (e),(e))
  static char __Compatible[100];
  //lint -esym(528,__Compatible)
  //lint -esym(551,__Compatible)
  //lint -esym(843,__Compatible)
#else
 #define Compatible(e,type) (e)
#endif
```

You could then define the original `Shift_Left` macro as:

```
#define Shift_Left(u,b) ((u) << compatible(b,Bits))
```

`Compatible(e,type)` works as follows. Under normal circumstances (i.e. when compiling) it is equivalent to the expression `e`. When linting it is also equivalent to `e` except that there is a side effect of assigning to some obscure array that has been artfully configured into resembling a data object of type `type`. A complaint will be issued if the expression `e` would draw a complaint when assigned to an object of type `type`.

In this way you can be assured that the shift amount is always assignment compatible with `Bits`. Note that there is no longer a need for the twin `Shift_Left` definitions. And `Compatible` can be used in many other places to assure that objects are typed according to program requirements.

For simplicity, we have focused on shifting left. Obviously, similar comments can be made for shifting right.

7.5 Strong Type Hierarchies

7.5.1 The Need for a Type Hierarchy

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

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

#define A_FLAG (Flags1) 1
#define B_FLAG (Flags2) 1
#define C_FLAG (Flags3) 1
```

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

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

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

7.5.2 The Natural Type Hierarchy

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

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

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

Hierarchy depends on the state of the `fhs` (Hierarchy of Strong types) flag, which is normally ON. If you turn it off with the

```
-fhs
```

option the natural hierarchy is not formed.

We say that `Flags` is a *parent* type to each of `Flags1`, `Flags2` and `Flags3`, which are its children. Being a parent to a child type is similar to being a base type to a derived type in an object-oriented system with one difference. A parent is normally interchangeable with each of its children; a parent can be assigned to a child and a child can be assigned to a parent. But a base type cannot normally be assigned to a derived type. But even this property can be obtained via the `-father` option (See Section 7.5.4 Restricting Down Assignments (-father)).

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

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

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

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

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

Type hierarchies can be an arbitrary number of levels deep.

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

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

7.5.3 Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via `typedef` declaration has a limitation. All the types in a single tree must be the same underlying type. The `-parent` option can be used to supplement (or completely replace) the strong type hierarchy established via typedef declarations.

An option of the form:

-parent(*Parent* , *Child* [, *Child*] ...)

where *Parent* and *Child* are type names defined via `typedef` will create a link in the strong type hierarchy between the *Parent* and each of the *Child* types. The *Parent* is considered to be equivalent to each *Child* for the purpose of Strong type matching. The types need not be the same underlying type and normal checking between the types is unchanged.

A link that would form a loop in the tree is not permitted.

For example, given the options:

```
-parent(Flags1,Small)
-strong(AJX)
```

and the following code:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef unsigned char Small;
```

then the following type hierarchy is established:

```
    Flags
     /   \
Flags1   Flags2
  |
Small
```

If an object of type `Small` is assigned to a variable of type `Flags1` or `Flags`, no strong type violation will be reported. Conversely, if an object of type `Flags` or `Flags1` is assigned to type `Small`, no strong type violation will be reported but a loss of precision message will still be issued (unless otherwise inhibited) because normal type checking is not suspended.

If the `-fhs` option is set (turning off the hierarchy of strong types flag) a `typedef` will not add a hierarchical link. The only links that will be formed will be via the `-parent` option.

7.5.4 Restricting Down Assignments (-father)

The option

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

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

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

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

Here `Index` is a special Index into an array. `FIndex` is a Flag or an `Index`. If negative, `FIndex` is taken to be a special flag and otherwise can take on any of the values of `Index`. By defining `Index` in terms of `FIndex` we are implying that `FIndex` is the parent of `Index`. The reader not accustomed to OOP may think that we have the derivation backwards, that the simpler `typedef`, `Index`, should be the parent. But `Index` is the more specific type; every `Index` is an `FIndex` but not conversely. Whereas it is expected that we can assign from `Index` to `FIndex` it could be dangerous to do the inverse.

Since we do not want down assignments we give the option

```
-father( FIndex, Index )
```

in addition to the strong options, say

```
-strong( AcJcX, FIndex, Index )
```

Then

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

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

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

```
Index F_to_I( FIndex fi )
```

7.6 Printing the Hierarchy Tree

To obtain a visual picture of the hierarchy tree, use the letter 'h' in connection with the -v option. For example, using the option +vhm for the example in Section 7.5.3 Adding to the Natural Hierarchy you will capture the following hierarchy tree.

```
--Flags
  |
  |
  +--Flags1
  |  |
  |  |__Small
  |
  |__Flags2
```

To get a more compressed tree (vertically) you may follow the 'h' with a '-'. This results in a tree where every other line is removed. For example, if you had used the option +vh-m the same tree would appear as:

```
--Flags
  |--Flags1
  |  |__Small
  |__Flags2
```

7.7 Reference Information

7.7.1 Full Source for the Gravitation Example

```
//lint -strong(JAc, Meter, Kilogram, Second)
//lint -strong(JAc, Area = Meter * Meter)
//lint -strong(JAc, Volume = Meter * Meter * Meter)
//lint -strong(JAc, Velocity = Meter / Second)
//lint -strong(JAc, Acceleration = Meter / (Second * Second))
//lint -strong(JAc, Newton = Kilogram * Acceleration)
/*lint -strong(JAc, GravitationalConstant =
               Newton * Area / (Kilogram * Kilogram)
              )
*/
typedef double Meter, Second, Velocity, Acceleration;
typedef double Kilogram, Newton;
typedef double Area, Volume;
typedef double GravitationalConstant;

const GravitationalConstant G = 6.67e-11;

Newton attraction(Kilogram mass1, Kilogram mass2, Meter distance) {
        return (mass1 * mass2) / (distance * distance);
}
```

7.7.2 The Strong Type of an Expression

An expression is strongly typed if:

1. It is a strongly typed variable or field.
2. It is the return value of a function whose return type is strong.
3. It is a cast to a strong type.
4. It is one of the type-propagating unary operators, `+`, `-`, `++`, `--`, and ~, applied to a strongly typed expression.
5. It is the result of dereferencing a pointer to a strong type, or of indexing an array.
6. It is the result of a multiplicative binary operator, `*` or `/`, where at least one operand is dimensional and the operands can be combined through dimensional analysis.
7. It is formed by one of the type-propagating binary operators where both operands are strongly typed expressions with compatible strong types. The type-propagating binary operators are the arithmetic operators, `+`, `-`, `*`, `/`, and `%`, the bitwise operators, `&`, `|`, and `^`, and the conditional operator, `?:`, where the two operands are the true and false arms.
8. It is a shift operator whose left operand is a strong type. The strong type of the shift operator is then the strong type of the left operand.
9. It is a comma operator whose right operand is a strong type. The strong type of the comma operator is then the strong type of the right operand.
10. It is an assignment operator whose left side is a strong type. The strong type of the assignment operator is then the strong type of the left operand.
11. A boolean strong type has been designated and it is a comparison operator, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`, `||`, and `&&`. It will have the strong type of the designated boolean strong type.

7.7.3 Canonical Form for Dimensional Strong Types

Every strong type is reduced to a canonical form internally. Dimensional strong types may be specified using any valid C expression containing:

binary `*` `/` operators
identifiers (including the special identifier `1` for numerators)
balanced parentheses

Output (in messages and the output of `-vh`) will always be presented in the canonical form where all terms are reduced, consecutively multiplied operands are sorted lexicographically, multiplicative expressions as operands to division are parenthesized, a missing numerator is replaced with a `1`, and a denominator of `1` is omitted (and its dividend not parenthesized).

7.7.4 Message Numbers

The primary message numbers related to Strong Types are: 18, 138, 463, 632, 633, 634, 635, 636, 637, 638, 639, 640, and 697. Setting a strong boolean type will affect the behavior of messages involving boolean contexts that are otherwise unrelated to strong types.

8 Value Tracking

8.1 Introduction

Most components of PC-Lint Plus analyze a static compile-time view of your code. This is sufficient for most static analysis tasks, but detection of certain runtime problems requires a deeper approach. Value Tracking combines static and dynamic analysis with an in-depth knowledge of C and C++ programming patterns to find potential runtime errors by performing an approximate symbolic interpretation of your code without the exponential slowdown incurred from a blind purely symbolic analysis. Fox example:

```
int g(int x) {
    x = x - 5;
    return 100 / x;
}

void f() {
    int a = 3;
    a += 2;
    a = g(a);
}
```

Value Tracking will find the division by zero error and explain how it occurs:

```
3 warning 414: possible division by zero
    return 100 / x;
               ^ ~
9 supplemental 894: during specific walk g(5)
    a = g(a);
        ^
2 supplemental 831: assignment yields 0
    x = x - 5;
    ~~^~~~~~~
2 supplemental 831: operator - yields 0
    x = x - 5;
        ~~^~~
1 supplemental 831: argument initialization yields 5
int g(int x) {
      ~~~~^
9 supplemental 831: argument passing yields 5
    a = g(a);
          ^
8 supplemental 831: operator + yields 5
    a += 2;
    ~~^~~~
```

8.1.1 Anatomy of a Value Tracking Message

The division by zero warning shown in the previous example consists of three parts. The primary message, warning 414, describes the problem that Value Tracking has detected. The next message in the message group is supplementary message 894, which is used to indicate that the primary message was issued during a specific walk of a function using arguments passed to it in a function call within an earlier walk. Message 894 may occur multiple times in a message group and indicates the values of function arguments on the specific call stack. The repeated instances of message 831 are used to show how the history of a relevant variable led to the occurrence of the situation that was reported as suspicious in the primary message, in reverse chronological order. The first instance of message 831 shows how `x` came to have the value `0` and the following messages proceed backwards to the value's origin.

8.2 Value Inferencing

8.2.1 Conditionals

Inside of a conditionally executed region, such as the body of an `if` statement, PC-lint Plus will infer that the condition required for the region to execute must be true when that region begins executing. In the case of an `if` statement, it will further infer that the opposite of such a condition must be true when a matching `else` statement begins executing. Inferred values and modifications made to variables within conditional regions generally do not survive once the conditional scope exits. If the condition can be determined to always be true or false under the circumstances, or one branch of an `if` statement always returns, then changes, inferences, or even a reversed inference may persist beyond the conditionally executed region. Inferencing can be disabled using the `ii` flag, although this is not recommended.

8.2.2 Assertions

Values can be inferred from expressions passed as arguments to functions using the appropriate assert semantic. Inferences derived from assertions work in the same manner as inferences derived from conditionals. The standard assert macro may be redefined as an invocation of function with the appropriate semantics if necessary for your standard library implementation. See 9.1.2 Semantics for more information. For example:

```
void __assert(bool); // this special function has the assert semantic by default
#define assert(x) __assert(x)

int x;

int* g() {
    if (x) return &x;
    else return nullptr;
}

void f() {
    int* a = g();
    assert(a);
    *a = 10; // no warning about potential use of null pointer due to assert
}
```

8.3 Integer Range Tracking

Rather than tracking only a single value, an integer can be represented by a range of possible values. For example:

```
void f(int a) {
    if (a > 5 && a < 10) {
        /* a is known to be between six and nine inclusive here */
    }
}

void g(unsigned a) {
    if (a > 10) return;
    /* a is known to be between zero and ten inclusive here */
}
```

You can test these examples using the integrated debugger as explained below in Section 8.7 Debugger.

8.4 Terminology

- *General walk* - The initial analysis of a function, independent of the rest of the program. The values of function arguments are completely unknown during a general walk. PC-Lint Plus generally will not report violations relating to these unknown arguments unless it has learned something to constrain their possible values. This helps to avoid false positives. Function calls encountered in the general walk launch specific walks.
- *Specific walk* - A walk of a function launched by a function call during a general walk or an earlier specific walk (up to the depth limit). The arguments passed at the call site are known during this walk.
- *Depth* - The number of nested specific walks at the current point in execution.
- *Pass* - A repeatable event consisting of the processing or reprocessing of each source file. Each file is read from the disk and analyzed again in each pass. Information stored between passes allows some messages to provide more information as the number of passes increases. Default operation involves two passes, one in which analysis local to each module is performed while globally relevant information is collected, and another in which global information collected in the previous pass is acted on for each module (Global Wrap-up). Global Wrap-up only occurs once, in the second pass, if Global Wrap-up is enabled at all. Value Tracking is performed again in each pass.

8.5 Value Display Format

8.5.1 Integers

The value of an integer will be printed in base ten. If there are a range of possible values, the minimum and maximum values (both inclusive) will be printed, separated by a colon. If the value of an integer is not known but there is reason to believe it may be zero, a zero followed by a question mark will be printed.

8.5.2 Pointers in General

If a pointer is null, the string `nullptr` will be printed. If a pointer may possibly be null, a question mark will be printed after the pointer's value. If a pointer is custodial, the value will be followed by a suffix of `C` or `c`, with the lowercase form indicating some degree of uncertainty. A pointer derived from the conversion of an integer literal specifying a fixed constant address will be indicated with a suffix of `F`.

8.5.3 Pointers to a Single Datum

A pointer to a unique object not considered to be part of an array or multi-element allocation will be printed as `&(V)` where `V` represents the value of the target object.

8.5.4 Pointers to Buffers with Multiple Elements

A pointer into a multi-element allocation will be printed as `[E]@I/S`. `E` represents the size of the allocation in bytes. `I` represents the index into the allocation in bytes. `E` and `I` may be displayed as ranges under the rules specified above for integers. `S` is the size of the element type. The suffix `NUL@Z` represents the belief that a null terminator is present at index `Z`, which may be a range.

8.5.5 Objects of Structure or Class Type

Members and base class sub-objects are printed in a comma separated list delimited by curly braces and prefixed with `.` or `:` respectively. For example, given these structures:

```
struct X {
    int a;
}

struct Y : X {
```

```
    int b;
};
```

an object of type `Y` may be displayed as `{ :X = { .a = 42 }, .b = 11 }`.

8.5.6 Uninitialized Values

Uninitialized values are represented using the string `uninitialized`. A question mark suffix may appear if a value is only possibly uninitialized.

8.6 General Usage

Value Tracking is enabled by default. More information can be revealed by increasing the specific walk depth using the `-vt_depth` option. Larger values will increase runtime and (peak) memory usage. A variety of more specific Value Tracking features can be controlled using 9.1.2 Semantics.

8.7 Debugger

A debugger in the spirit of `gdb` is provided to probe the state of the Value Tracking interpreter during execution. This functionality can be accessed using the `+fvd`. The flag can be enabled in a lint comment to avoid triggering the debugger earlier in the program. For example, given `a.cpp`:

```
void f(int a) {
    int b = 5;
    a = b + 2;
    int c = a;
}
```

running with the debugger enabled will present the following modal interface:

```
    @ a.cpp:2
    ^
    # void f(int a) {
--> #     int b = 5;
    #     a = b + 2;
    #     int c = a;
(vt)
```

which indicates that the debugger has stopped prior to the execution of the line indicated by the arrow in the left margin and listed after the filename in the header. Pressing enter without inputting any text at the `(vt)` prompt will proceed to the next statement.

```
    @ a.cpp:3
    # void f(int a) {
    #     int b = 5;
--> #     a = b + 2;
    #     int c = a;
    # }
```

Entering `?` will print the current values of all variables in scope:

```
### unknown storage ###
### static storage ###
### dynamic storage ###
### function f parameters ###
a = unknown $1
### compound statement ###
b = 5 $3
```

Each variable belongs to the scope (denoted with triple hash headings) that it appears immediately under. The variable `b` is local to the compound statement of the function body, and has the value `5`. The `$3` following the value represents the simulated memory slot in which the value of `b` resides. If the program is stepped to the end of the function, it will stop on the closing brace to provide a final opportunity to issue commands before leaving the scope.

More advanced features are available and listed in the output of the `help` command at the prompt. The debugger is considered experimental and subject to change. Value Tracking debugging is not available when using the Parallel Analysis feature.

8.8 Interfunction and Intermodule Value Tracking

Interfunction Value Tracking operates differently depending on whether a given function call is connecting two functions within the same module or two functions within different modules. PC-lint Plus can track values across an arbitrary number of intramodule function calls limited only by the value of the `-vt_depth` option and the amount of time available. The order in which functions are defined within a single module does not influence Value Tracking. While intramodule calls are processed depth-first, intermodule calls are processed breadth-first and the boundary from a given module to any other module can only be crossed in the next pass. The initial processing of each source module constitutes a single pass and an additional pass to utilize the information stored during this pass before the first intermodule results are produced. Another pass will occur by default as part of Global Wrap-up processing. Additional passes can be requested using the `-vt_passes` option.

Calls to library functions will not be walked unless the `flf` flag has been set to 2.

8.9 Limitations

8.9.1 Initial Values of Static Variables

The initial values of non-`const` static duration variables are not currently considered during Value Tracking. Changes to static variables within a function or call chain *are* tracked. For example:

```
int a;
int b;

int f() {
    return 5 / a;
}

int g() {
    b = 0;
    return 5 / b;
}
```

In the general walk, PC-lint Plus will report on the division by zero in `g` but will not report on the division by `a` in `f` because while `a` is initialized to `0`, there is no information in the function to suggest that `a`, which could have been modified by another function in the program, has any particular value.

8.9.2 The Correlated Variables Problem

Correlations between independent variables are not currently tracked. In this example:

```
void f(int* p) {
    int* a = 0;
    if (p) { a = p; }
    bool is_null = !p;
    if (!is_null) {
```

```
6         *a = 10;
7     }
8 }
```

PC-lint Plus will report the potential for a null pointer dereference on line 6 while this is technically not possible because the value of `is_null` is correlated with the value of `a`. Nonetheless, reports under these circumstances can be useful because use of this pattern can lead to brittle code. This can be remedied by placing an assertion before line 6, such as `assert(a);`, which will prevent the warning.

8.9.3 Terminal Depth Assistance

PC-lint Plus will make an exception to the depth limit for a call to a:

- `constexpr` function
- literal operator
- `const` member function returning `bool`

This helps avoid unexpected results in certain cases where the depth limit would otherwise need to be increased. This is not a substitute for specifying the optimal depth for your desired analysis using the `-vt_depth` option.

8.10 Changes from Older Products

- There is no longer a distinction between static and dynamic messages for suppression purposes because the new architecture of PC-lint Plus does not run non-dynamic checks multiple times.
- Value Tracking is now performed depth-first, the same way your program executes on a real machine. Previous products performed a breadth-first search due to the multiple pass architecture used. This implies the removal of `-static_depth`, which is now effectively infinite.
- Integer tracking can now track a range of possible values.
- Floating point values, pointer targets, function pointers, and structure members can now be tracked.
- The reference information format has changed. An example of the new, more detailed reference information is provided in the introduction.
- The "conceivable" severity for conditions dependent on a loop never being entered has been retired.

9 Semantics

9.1 Function Mimicry (-function)

This section describes how some properties of built-in functions can be transferred to user-defined functions by means of the option `-function`. See also `-printf` and `-scanf`. See also Section 9.2 Semantic Specification to see how to create custom function semantics.

9.1.1 Special Functions

PC-lint Plus is aware of the properties (which we will call semantics) of many standard functions, which we refer to as special functions. A complete list of such functions is shown in Section 9.1.2 Function Listing.

For example, function `fopen()` is recognized as a special function. Its two arguments are checked for the possibility of being the NULL pointer and its return value is considered possibly NULL. Similarly, `fclose` is regarded as a special function whose one argument is also checked for NULL. Thus, the code:

```
if( name ) printf ( "ok\n" );
f = fopen( name, "r" );        // Warning! name may be NULL
fclose ( f );                  // Warning! f may be NULL
```

will be greeted with the diagnostics indicated. You may transfer all three semantics of `fopen` to a function of your own, say `myopen`, by using the option

```
-function( fopen, myopen )
```

Then, PC-lint Plus would also check the 1st and 2nd arguments of `myopen` for NULL and assume that the returned pointer could possibly be NULL. In general, the syntax of `-function` is described as follows:

```
-function( Function0, Function1 [, Funcion2] ... )
```

specifies that *Function1*, *Function2*, etc. are like *Function0* in that they exhibit special properties normally associated with *Function0*.

The arguments to `-function` may be subscripted. For example, if `myopen` were to check its 2nd and 3rd arguments for NULL rather than its 1st and 2nd we could do the following:

```
-function( fopen(1), myopen(2) )
-function( fopen(2), myopen(3) )
```

This would transfer the semantics of NULL checking to the 2nd and 3rd arguments of `myopen`. This could be simplified to

```
-function( fopen(1), myopen(2), myopen(3) )
```

since the property of `fopen(1)` is identical to that of `fopen(2)`. Any previous semantics associated with the 2nd and 3rd arguments to `myopen` would be lost. To transfer the return semantics you may use the option

```
-function( fopen(r), myopen(r) )
```

Some functions have a semantic that is not decomposable to a single argument or return value but is rather a combined property of the entire function. For example

```
char * fread( char *, size\_{t}, size\_{t}, FILE * );
```

has, in addition to the check-for-NULL semantics on its 1st and 4th arguments, and the check-for-negative semantics on the 2nd and 3rd arguments, an additional check to see if the size of argument 2 multiplied by argument 3 exceeds the buffer size given as the 1st argument. This condition is identified as semantic `fread` in Section 9.1.2 Function Listing. Thus

```
char buf[100];
fread( buf, 100, 2, f );       // Warning
```

To transfer this function-wide property to some other function we need to use the 0 (zero) index. Thus

```
-function( fread(0), myread(0) )
```

will transfer just the overflow checking (`fread` as described above) and not the argument checking. That is, of the semantics appearing in Section 9.1.2 Function Listing for row labeled `fread`, the semantics transferred are only those marked with and asterisk.

As a convenience, the subscript need not be repeated if it is the same as the 1st argument. Thus

```
-function( fread(0), myread )
```

is equivalent to the earlier option.

Just as in the case of `fopen` you may transfer all the properties of `fread` to your own function by not using a subscript as in:

```
function( fread, myread )
```

You may remove any or all of these semantics from a special function by not using a 2nd argument to the `-function` option. Thus

```
-function( fread )
```

will remove all of the semantics of the `fread` function and

```
-function( fread(0) )
```

removes only the special semantics described above.

In summary, an option of the form

-function(*function*(*index*), ...)

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

-function(*function*, ...)

copies all of a function's semantics.

You may transfer semantics to member functions as well as non-member functions. Thus

```
-function( exit, X::quit )
```

transfers the properties of `exit()` to `X::quit()`. The semantics in this case is simply that the function is not expected to return.

As another example involving member functions consider the following:

```
//lint -function( strlen(1), X::X(1), X::g )
// both X::X() and X::g() should have their 1st
// argument checked for NULL.
//lint +fpn pointer parameters may be NULL

class X {
public:
    char *buf;
    X(char *);
    void g(char *);
};

void f(char *p) {    // p may be NULL because of +fpn
    X x(p);          // Warning 668
    x.g(p);
}
```

In this example, the semantics associated with the 1st argument of `strlen` are transferred to the 1st argument of `X::X` and to the 1st argument of `X::g`. As the example illustrates, when we speak of the *n*th argument passed to a member function we are ignoring in our count the implicit argument that is a pointer to the class (this is always checked for NULL).

No distinction is made among overloaded functions. Thus, if `X::X( int * )` is checked for NULL then so is `X::X( char * )`. If there is an `X::X( int )` then its argument is not checked because its argument is not a pointer. If there is an `X::X( int *, char * )` then the 1st argument is checked, but not the 2nd. User-defined semantics can be applied to individual function overloads or function template instantiations, see Section 9.2 Semantic Specifications for details.

9.1.2 Function Listing

Function	Semantics
`_Exit`	r_no
`__assert`	**assert*
`abort`	r_no
`acos`	**dom_1*
`acosf`	**dom_1*
`acosh`	**dom_lt1*
`acoshf`	**dom_lt1*
`acoshl`	**dom_lt1*
`acosl`	**dom_1*
`asctime`	1p
`asctime_s`	1p chneg(2) 3p
`asin`	**dom_1*
`asinf`	**dom_1*
`asinl`	**dom_1*
`at_quick_exit`	1p
`atanh`	**dom_1*
`atanhf`	**dom_1*
`atanhl`	**dom_1*
`atexit`	1p
`atof`	1p
`atoi`	1p
`atol`	1p
`atoll`	1p
`bsearch`	1p 2p chneg(3) chneg(4) 5p r_null
`bsearch_s`	chneg(3) chneg(4)
`call_once`	1p 2p
`calloc`	chneg(1) chneg(2) r_null **calloc*
`clearerr`	1p
`cnd_broadcast`	1p
`cnd_destroy`	1p
`cnd_init`	1p
`cnd_signal`	1p
`cnd_timedwait`	1p 2p 3p
`cnd_wait`	1p 2p
`ctime`	1p

Function	**Semantics**
`ctime_s`	1p chneg(2) 3p
`exit`	r_no
`fclose`	1p **fclose*
`feof`	1p
`ferror`	1p
`fflush`	1p
`fgetc`	1p
`fgetpos`	1p 2p
`fgets`	1p chneg(1) 3p r_null **fgets*
`fopen`	1p 2p r_null
`fopen_s`	1p 2p 3p
`fprintf`	1p 2p printf(2)
`fprintf_s`	1p 2p printf(2)
`fputc`	2p
`fputs`	1p 2p
`fread`	1p chneg(2) chneg(3) 4p **fread*
`free`	**free*
`freopen`	2p 3p r_null
`freopen_s`	1p 3p 4p
`frexp`	2p
`frexpf`	2p
`fscanf`	1p 2p scanf(2)
`fscanf_s`	1p 2p scanf(2)
`fseek`	1p
`fsetpos`	1p 2p
`ftell`	1p
`fwprintf`	1p 2p printf(2)
`fwprintf_s`	1p 2p printf(2)
`fwrite`	1p chneg(2) chneg(3) 4p **fwrite*
`fwscanf`	1p 2p scanf(2)
`fwscanf_s`	1p 2p scanf(2)
`getc`	1p
`getenv`	1p
`getenv_s`	1p chneg(3)
`gets`	1p dangerous r_null
`gets_s`	1p chneg(2)
`gmtime`	1p r_null
`gmtime_s`	1p 2p
`localtime`	1p r_null
`localtime_s`	1p 2p
`log10`	**dom_lt0*
`log10f`	**dom_lt0*
`log10l`	**dom_lt0*
`log1p`	**dom_lt1*
`log1pf`	**dom_lt1*
`log1pl`	**dom_lt1*
`log2`	**dom_lt0*
`log2f`	**dom_lt0*

Function	Semantics
`log2l`	**dom_lt0*
`longjmp`	1p r_no
`malloc`	chneg(1) r_null **malloc*
`mbsrtowcs_s`	1p chneg(3) 4p chneg(5) 6p
`mbstowcs`	1p 2p chneg(3)
`mbstowcs_s`	1p chneg(3) 4p chneg(5)
`memchr`	1p pod(1) chneg(3) r_null **memchr*
`memcmp`	1p pod(1) 2p pod(2) chneg(3) **memcmp*
`memcpy`	1p pod(1) 2p pod(2) chneg(3) **memcpy*
`memcpy_s`	1p pod(1) chneg(2) 3p pod(3) chneg(4)
`memmove`	1p pod(1) 2p pod(2) chneg(3) **memcpy*
`memmove_s`	1p pod(1) chneg(2) 3p pod(3) chneg(4)
`memset`	1p pod(1) chneg(3) **memset*
`memset_s`	1p pod(1) chneg(2) chneg(3)
`mktime`	1p inout(1)
`modf`	2p
`modff`	2p
`modfl`	2p
`mtx_destroy`	1p
`mtx_init`	1p
`mtx_lock`	1p
`mtx_timedlock`	1p 2p
`mtx_trylock`	1p
`mtx_unlock`	1p
`perror`	1p
`printf`	1p printf(1)
`printf_s`	1p printf(1)
`putc`	2p
`puts`	1p
`qsort`	1p inout(1) chneg(2) chneg(3) 4p
`qsort_s`	inout(1) chneg(2) chneg(3)
`quick_exit`	r_no
`realloc`	r_null **realloc*
`remove`	1p
`rename`	1p 2p
`rewind`	1p
`scanf`	1p scanf(1)
`scanf_s`	1p scanf(1)
`setbuf`	1p
`setvbuf`	1p
`snprintf`	chneg(2) 3p printf(3) **sprintf*
`snprintf_s`	1p chneg(2) 3p printf(3) **sprintf*
`snwprintf_s`	1p chneg(2) 3p printf(3)
`sprintf`	1p 2p printf(2) **sprintf*
`sprintf_s`	1p chneg(2) 3p printf(3) **sprintf*
`sqrt`	**dom_lt0*
`sqrtf`	**dom_lt0*
`sqrtl`	**dom_lt0*

Function	Semantics
`sscanf`	1p 2p scanf(2)
`sscanf_s`	1p 2p scanf(2)
`strcat`	1p inout(1) 2p **strcat*
`strcat_s`	1p inout(1) chneg(2) 3p
`strchr`	1p type(1) r_null
`strcmp`	1p 2p
`strcoll`	1p 2p
`strcpy`	1p 2p **strcpy*
`strcpy_s`	1p chneg(2) 3p
`strcspn`	1p 2p
`strerror_s`	1p chneg(2)
`strftime`	1p chneg(2) 3p 4p
`strlen`	1p **strlen*
`strncat`	1p inout(1) 2p chneg(3) **strncat*
`strncat_s`	1p inout(1) chneg(2) 3p chneg(4)
`strncmp`	1p 2p chneg(3)
`strncpy`	1p 2p chneg(3) **strncpy*
`strncpy_s`	1p chneg(2) 3p chneg(4)
`strpbrk`	1p type(1) 2p r_null
`strrchr`	1p type(1) r_null
`strspn`	1p 2p
`strstr`	1p type(1) 2p r_null
`strtod`	1p
`strtof`	1p
`strtok`	inout(1) 2p r_null
`strtok_s`	inout(1) 2p 3p 4p
`strtol`	1p
`strtold`	1p
`strtoll`	1p
`strtoul`	1p
`strtoull`	1p
`strxfrm`	2p chneg(3)
`swprintf`	1p chneg(2) 3p printf(3)
`swprintf_s`	1p chneg(2) 3p printf(3)
`swscanf`	1p 2p scanf(2)
`swscanf_s`	1p 2p scanf(2)
`thrd_create`	1p
`thrd_sleep`	1p
`timespec_get`	1p
`tmpfile`	r_null
`tmpfile_s`	1p
`tmpnam_s`	1p chneg(2)
`tss_create`	1p
`ungetc`	2p
`vfprintf`	1p 2p 3p printf(2)
`vfprintf_s`	1p 2p 3p printf(2)
`vfscanf`	1p 2p 3p scanf(2)
`vfscanf_s`	1p 2p 3p scanf(2)

Function	Semantics
vfwprintf_s	1p 2p 3p printf(2)
vfwscanf_s	1p 2p 3p scanf(2)
vprintf	1p 2p printf(1)
vprintf_s	1p 2p printf(1)
vscanf	1p 2p scanf(1)
vscanf_s	1p 2p scanf(1)
vsnprintf	chneg(2) 3p 4p printf(3)
vsnprintf_s	1p chneg(2) 3p 4p printf(3)
vsnwprintf_s	1p chneg(2) 3p 4p printf(3)
vsprintf	1p 2p 3p printf(2)
vsprintf_s	1p chneg(2) 3p 4p printf(3)
vsscanf	1p 2p 3p scanf(2)
vsscanf_s	1p 2p 3p scanf(2)
vswprintf_s	1p chneg(2) 3p 4p printf(3)
vswscanf_s	1p 2p 3p scanf(2)
vwprintf_s	1p 2p printf(1)
vwscanf_s	1p 2p scanf(1)
wcrtomb_s	1p chneg(3) 5p
wcscat_s	1p chneg(2) 3p
wcscpy_s	1p chneg(2) 3p
wcsncat_s	1p chneg(2) 3p chneg(4)
wcsncpy_s	1p chneg(2) 3p chneg(4)
wcsrtombs_s	1p chneg(3) 4p chneg(5) 6p
wcstok_s	2p 3p 4p
wcstombs	1p 2p chneg(3)
wctomb	1p
wmemcpy_s	1p chneg(2) 3p chneg(4)
wmemmove_s	1p chneg(2) 3p chneg(4)
wprintf	1p printf(1)
wprintf_s	1p printf(1)
wscanf	1p 2p scanf(2)
wscanf_s	1p scanf(1)

Semantics

assert The function argument can be assumed to be true (non-zero).

calloc The length of the returned buffer is the product of the first and second arguments or the returned pointer is NULL.

dom_1 The specified argument(s) must be in the range of [-1, 1] as a value outside this range is not defined for this function and may result in a domain error; violations will be diagnosed with message 2423/2623.

dom_lt1 The specified argument(s) must not be less than 1 as such a value is not defined for this function and may result in a domain error; violations will be diagnosed with message 2423/2623.

dom_lt0 The specified argument(s) must not be less than 0 as such a value is not defined for this function and may result in a domain error; violations will be diagnosed with message 2423/2623.

exit The function never returns.

fclose Pointer argument 1 is regarded as being uninitialized after the function returns.

`fgets` Integer argument 2 should not exceed the size of the buffer pointed to by argument 1; violations will be diagnosed with message 419/669 (data overrun).

`fread` The product of the integer arguments 2 and 3 should not exceed the size of buffer argument 1; violations will be diagnosed with message 419/669 (data overrun).

`free` Pointer argument 1 is regarded as being uninitialized after the function returns and the pointed to memory is marked as having been freed (attempting to free the same memory a second time will be diagnosed by message 449). Additionally, if the memory pointed to by argument 1 was derived from an allocation source not appropriate for deallocation via the `free` function, this will be diagnosed via message 424.

`fwrite` The product of the integer arguments 2 and 3 should not exceed the size of buffer argument 1; violations will be diagnosed with message 420/670 (access beyond end of array).

`malloc` The length of the buffer returned is the value of integer argument 1 or the returned pointer is NULL.

`memchr` Integer argument 3 should not be larger than the size of the buffer pointed to by argument 1; violations will be diagnosed with message 420/670 (access beyond end of array).

`memcmp` Integer argument 3 should not be larger than the size of the buffer pointed to by either argument 1 or argument 2; violations will be diagnosed with message 420/670 (access beyond end of array).

`memcpy` Integer argument 3 should not be larger than the size of the buffer pointed to by either argument 1 or argument 2; a value that exceeds argument 1 will be diagnosed with message 419/669 (data overrun), a value that exceeds argument 2 will be diagnosed with message 420/670 (access beyond end of array).

`memset` Integer argument 3 should not be larger than the size of the buffer pointed to by argument 1; violations will be diagnosed with message 419/669 (data overrun).

`sprintf` Message 464 will be issued if the call to this `sprintf`-like function will result in the destination string being written onto itself.

`reall_1` Pointer argument 1 is regarded as being possibly uninitialized after the function returns.

`realloc` The length of the buffer returned is the value of integer argument 1 or the returned pointer is NULL.

`strcat` The size of buffer argument 2 should not be larger than the size of buffer argument 1; violations will be diagnosed with message 419/669 (data overrun).

`strcpy` The size of buffer argument 2 should not be larger than the size of buffer argument 1; violations will be diagnosed with message 419/669 (data overrun).

`strncat` Integer argument 3 should not be larger than the size of buffer argument 1; violations will be diagnosed with message 419/669 (data overrun).

`strncpy` Integer argument 3 should not be larger than the size of buffer argument 1; violations will be diagnosed with message 419/669 (data overrun).

9.2 Semantic Specifications (-sem)

The `-sem()` option allows the user to endow his functions with user-defined semantics. This may be considered an extension of the `-function()` option (See Section 9.1 Function Mimicry (-function)). Recall that with the `-function()` option the user may copy the semantics of a built-in function to any other function but new semantics cannot be created.

With the `-sem` option, entirely new checks can be created; integral and pointer arguments can be checked in combination with each other using usual C operators and syntax. Also, you can specify some constraints upon the return value.

The format of the `-sem()` option is:

```
-sem( function [,sem] ...)
```

This associates the semantics *sem* ... with the named function *function*. The semantics *sem* are defined below. If no *sem* is given, i.e. if only *function* is given, the option is taken as a request to remove semantics from the named function. Once semantics have been given to a named function, the `-function()` option may be used to copy the semantics in whole or in part to other functions.

9.2.1 Possible Semantics

sem may be one of:

`r_null` the function may return the null pointer.

This information is used in subsequent value tracking. For example:

```
/*lint -sem( f, r_null ) */
char *f();
char *p = f();
*p = 0; /* warning, p may be null */
```

This is the same semantic that is employed for the builtin function semantics such as `bsearch`, `calloc`, and `fgets` in Section 9.1.2 Function Listing, and it is considered a Return semantic. See Section 9.1 Function Mimicry (-function) for the definition of Return semantic. A more flexible way to provide Return semantics is given below under expressions (*exp*).

`r_no` the function does not return.

Code following such a function is considered unreachable. This semantic is identical to the semantic used for the `exit()` function as shown in Section 9.1.2 Function Listing. This also is considered a Return semantic.

i`p` (e.g. 3p) the *i*th argument should be checked for null.

If the *i*th argument could possibly be null this will be reported. For example:

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

`initializer`

Some member functions are used to initialize members. They may be called from constructors or called directly when the programmer wants to reset the state of a class to what it would have been immediately after construction. In most cases, PC-lint Plus can automatically determine when such a function initializes class state, even if the initializing function calls other functions to perform parts of the initialization. When the body of the initializer function is not available to PC-lint Plus, such a determination cannot be made. In such cases, you may designate the member as an initializer using the `-sem` option. (The `initializer` semantic is a flag semantic). If a member is designated as an initializer function and the body is available to PC-lint Plus, a complaint will be issued if it fails to initialize all of the data members.

`cleanup`

The `cleanup` semantic does for destructors what `initializer` does for constructors. A function designated as `cleanup` is expected to process each (non-static) member pointer by either freeing it (in any of the various ways of releasing storage) or, at least, zeroing it. Failure to do this will merit Warning 1578. A function that is a candidate for this semantic will be pointed out by Warning 1579. `cleanup` is a flag semantic.

`inout(`*i*`)`

A semantic expression of the form `inout(`*i*`)` where *i* is a constant designating a parameter, indicates that an indirect object passed to that parameter will be both read and written by the function. Thus the *i*th parameter must be either a pointer (or, equivalently an array) or a reference.

This should not be used with pointers or references to `const` objects, since, in this case, it is assumed that the object referenced is only read by the function. It is considered an `in` parameter. If the parameter is a pointer or reference to a non-`const` it is assumed by default to be an out parameter. That is, the function will only write to the referenced object but will not read from it.

But there is no linguistic way to deduce that the argument will be both read and written such as, for example, the first argument to `strcat()`. Hence the need for this semantic.

For example:

```
//lint -sem( addto, inout(1) )

void addto( int *p, int b );   // add b to the object pointed to
                               // by the first argument.
void f() {
    int n;
    addto( &n, 12 );           // Warning, n is likely uninitialized
}
```

`custodial(`*i*`)` where *i* is some integer denoting the *i*th argument or the letter 't' denoting the `this` pointer.

It indicates that a called function will take 'custody' of a pointer passed to argument *i*. More accurately, it removes the burden of custody from its caller. For example,

```
//lint -sem(push,custodial(1))
void f() {
    int *p = new int;
    push(p);
}
```

Function `f` would normally draw a complaint (Warning 429) that custodial pointer `p` had not been freed or returned. However, with the custodial semantic applied to the first argument of `push`, the call to `push` removes from `f` the responsibility of disposing of the storage allocated to `p`.

To identify the implicit argument of a (non-static) member function you may use the 't' subscript. Thus:

```
//lint -sem( A::push, custodial(t) )
struct A { void push(); ... };
void g( ) {
    A *p = new A;
    p->push();
}
```

You can combine the custodial semantic with a test for NULL. For example,

```
-sem( push, 1p, custodial(1) )
```

will complain about NULL pointers being passed as first argument to push as well as giving the custodial property to this argument.

The custodial semantic is an argument semantic meaning that it can be passed on to another function using the argument number as subscript. Thus:

```
function( push(1), append(1) )
```

transfers the custodial property of the 1st argument of push (as well as the test for NULL) on to the 1st argument of function append. But note you may not transfer this semantics using a 0 subscript as that refers to function wide semantics.

An example of the use of the letter t to report this is as follows

```
/lint -sem( A::push, custodial(t) )
struct A { void push(); ... };
void g( ) {
      A *p = new A;
      p->push();
}
```

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

pod(*i*) A semantic expression of the form pod(*i*) where *i* is a constant designating a parameter, indicates that the argument is expected to be a pointer to a POD.

A POD is an abbreviation for Plain Old Datatype. In brief, an object of POD can be treated as so many bytes, copyable by memcpy, clearable by memset, etc. For example:

```
//lint -sem( clear, 1p, pod(1) ) wants a non-null pointer to POD
class A
    { A(); int data; } a;
class B
    { public: int data; } b;
void clear( void *, size_t );
void f() {
    clear( &a, sizeof(a) );    // Warning
    clear( &b, sizeof(b) );    // no Warning
}
```

pure This semantic will designate a function as being pure (see definition below).

Normally functions are determined to be pure or impure automatically through an analysis of their definition. However, if a function is external to the source files being linted, this analysis cannot be made and the function is by default considered impure. This semantic can be used to reverse this assumption so that the function is regarded as pure.

The significance of a pure function is that it lacks internal side-effects and this can be used to diagnose code redundancies. There are a number of places in the language (left hand side of a comma, first or third expression of a for clause, the expression statement) when it makes no sense to have an expression unless some side-effect is to be achieved. As an example

```
void f() {}
void g()
     {
     f();   // Warning 522
     }
```

Because we can deduce `f` to be pure, a warning is issued. In general, we may not be aware until pass 1 is finished that a function is pure. You can use the pure semantic to hasten the process of detection.

Another use of this semantic can be to determine on what grounds PC-lint Plus considers a function to be pure. If a function is designated as being pure and is later deemed to have impure properties Warning 453 will be issued with a detailed explanation as to why the function is impure.

Definition of a pure function: A function is said to be pure if it is not impure. A function is said to be impure if it modifies a static or global variable or accesses a volatile variable or contains any I/O operation, or makes a call to any impure function.

A function call is said to have side-effects if it is a call to an impure function or if it is a call to a pure function that modifies its arguments.
Example:

```
int n;
void e1()  { n++; }
void e2()  { static k; k++ }
void e3()  { printf ( "hello" ); }
double e4( double x )
           { return sqrt(x); }
void e5( volatile int k ) { k++; }
void e6()  {e1(); }
```

Each of the functions `e1` through `e6` is impure because it satisfies one of the above conditions of being an impure function. (This assumes that both `printf` and `sqrt` are external functions.) On the other hand, in the following:

```
int f1() { int n = 0; n++; return n; }
void f2( int*p ) { *p = f1(); }
```

both `f1` and `f2` are pure functions because there is nothing to designate them impure.

Consider:

```
//lint -sem( sqrt, pure )
void compute()
    {
    double x = sqrt( 2.0 );
    }
void m()
    { compute(); }
```

Here, because of the `pure` semantic given to `sqrt`, we get a deserved diagnostic (522, Highest operation, function '`compute`', lacks side-effects) at the call to `compute`. I'm sure the reader will agree that the function `compute` shows evidence of a lack of completeness. The author may have been side-tracked during development and never got back to completing the function. But as we indicated earlier `sqrt` would by default be considered impure since it is external. It may actually be impure since on error conditions it needs to set the external variable `errno` to `EDOM`.

Nonetheless, from the standpoint of desired functionability, `compute` comes up short. This can be traced to `sqrt` not offering any desired functionality as a side-effect. Since this is the case, the programmer was justified in inserting the semantic for `sqrt`.

Consider the following example:

```
int f()
    {
```

```
int n = 0;
n++;
return n;
}
```

`f()` is considered to be a pure function. True it modifies `n` but `n` is an automatic variable. The increment operator is not considered impure but it is regarded as having side-effects.

Consider the following pair of functions:

```
void h(int *p) { (*p)++; }
int g() { int n=0; h(&n); return n;}
```

Here the function `h()` is considered pure but note that the call `h(&n)` has side-effects. Function `g()` is exactly analogous to `f()` above and so must be considered pure. Function `g()` calls upon `h()` to modify variable `n` in much the same way that `f()` earlier employed the increment operator. If `g()` had provided the address of a global variable to `h()` then `g()` would have been considered impure but not `h()`. Had we considered `h()` to be impure irregardless of the nature of its argument then, since `g()` is pure, we would have had to give up the principle that impurity is inherited up the call chain.

`chneg(`*i*`)` A semantic expression indicating that the *i*th argument is expected to be non-negative.

Calling the function with a negative or possibly negative value will be diagnosed with message **422** or **671**, as appropriate.

`dangerous`
A function designated with the dangerous semantic will cause message **421** to be issued when the function is called. This is similar to function deprecation 11.8 Deprecation of Entities but using a semantic allows specific function overloads to be specified.

`printf(`*i*`)`
A semantic expression indicating that this is a `printf`-like function whose format argument is the *i*th argument. An option of the form `-sem(func, printf(i))` is functionally equivalent to the option `-printf(i,func)`.

`scanf(`*i*`)`
A semantic expression indicating that this is a `scanf`-like function whose format argument is the *i*th argument. An option of the form `-sem(func, scanf(i))` is functionally equivalent to the option `-scanf(i,func)`.

exp a semantic expression involving the expression elements described below:

- *i*`n` denotes the *i*th argument, which must be integral (E.g. `3n` refers to the 3rd argument). An argument is integral if it is typed `int` or some variation of integral such as `char`, `unsigned long`, an enumeration, etc.
- *i* may be `@` (commercial at) in which case the return value is implied. For example, the expression:

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

  states that the return value will either be equal to 4 or will be greater than the first argument.
- *i*`p` denotes the *i*th argument, which must be some form of pointer (or array). The value of this variable is the number of items pointed to by the pointer (or in the array). For example, the expression:

  ```
  2p == 10
  ```

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

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

- *i*`P` is like *i*`p` except that all values are specified in bytes. For example, the semantic:

  ```
  2P == 10
  ```

  specifies that the size in bytes of the area pointed to by the 2nd argument is 10. To specify a return pointer where the area pointed to is measured in bytes we use `@P`.
- *integer* (any C/C++ integral or character constant) denotes itself.
- *identifier* that refers to a macro that evaluates to a constant expression. The identifier is retained at option processing time and evaluated at the time of function call.
- `malloc(` *exp*`)` attaches a `malloc` allocation flag to the expression. See the discussion of Return Semantics below.
- `new(` *exp* `)` attaches a `new` allocation flag to the expression.
- `new[](` *exp* `)` attaches a `new[]` allocation flag to the expression.
- `( )`
- Unary operators: `+ - ! ~`
- Binary operators:

  `+ - * / % < <= == != > >= | & ^ < < >> || &&`
- Ternary operator: `?:`

9.2.2 Semantic Expressions

Operators, parentheses and constants have their usual C/C++ meaning. Also the precedence of operators are identical to C/C++.

There may be at most two expressions in any one -sem option, one expressing Return semantics and one expressing Function-wide semantics.

9.2.2.1 Return Semantics

An expression involving the return value (one of `@n`, `@p`, `@P`) is a Return semantic and indicates something about the returned value. For example, if the semantics for `strlen()` were given explicitly, they might be given as:

```
-sem( strlen, @n < 1p, 1p )
```

In words, the return value is strictly less than the size of the buffer given as first argument. Also the first argument should not be null.

To express further uncertainty about the return value, one or more expressions involving the return value may be alternated using the `||` operator. For example:

```
-sem( fgets, @p == 1p || @p == 0 )
```

represents a possible Return semantic for the built-in function `fgets`. Recall that the `fgets` function returns the address of the buffer passed as first argument unless an end of file (or error) occurs in which case the null pointer is returned. If the Return semantic indicates, in the case of a pointer, that the return value may possibly be zero (by explicitly using either the test `@p == 0` or `@P == 0` as in this example) this is taken as a possibility of returning the null pointer.

As another example:

```
-sem( lookup, 2n == LOCATE ? (@p==0||@p==1) : @p==1)
```

This is a Return semantic that says that if the 2nd argument of the function lookup is equal to `LOCATE` then the return pointer may or may not be null. Otherwise we may assume that the return value is a valid non-null pointer. This could be used as follows:

```
#define LOCATE 1
#define INSTALL 2
Symbol *lookup (const char *, int );
 ...
    sym = lookup("main", INSTALL);
    sym->value=0; /*OK*/
    sym = lookup("help", LOCATE);
    v = sym -> value; /* warning - could be NULL */
```

Here the first return value from lookup is guaranteed to be non-null, whereas the second may be null or may not be.

We caution the reader that the following, apparently equivalent, semantic does not work.

```
-sem( lookup, @p == (2n != LOCATE) || @p == 1 )
```

The OR (`||`) is taken to mean that either side or both could be true with some probability but there is no certainty deduced that one or the other must be true.

When `@p` (lowercase p) is used, the pointee type of the return value must be a complete type at the point the function is called since PC-lint Plus needs to be able to calculate the size of the returned buffer to use such a semantic. If the type return type is not complete, a 686 message will be issued.

Flagging the return value

Consider the example:

```
char *p, *q = 0;
p = malloc(10);
p = q;           // Warning -- memory leak
```

We are able to issue a Warning because the return from `malloc` has an allocation flag that indicated that the returned value points to a freshly allocated region that is not going to be freed by itself.

The seemingly equivalent semantic option:

```
-sem( my_alloc, @P == 1n )
```

associates no allocation flag with the returned pointer (only the size of the area in bytes).

To identify the kind of storage that a function may return, three flag-endowing functions have been added to the allowed expression syntax of the `-sem` option:

a region allocated by `malloc` to be released through `free`

a region allocated by `new` to be released through `delete`

a region allocated by `new[]` to be released through `delete[]`

In each case, the `exp` is the size of the area to be allocated. For example, to simulate `malloc` we may have:

```
-sem( my_alloc, @P == malloc(1n) )
```

By contrast the semantic:

```
-sem( some_alloc, @p == malloc(1n) )
```

indicates, because of the lower case 'p', that the size of the allocated region is measured in allocation units. Thus the `malloc` here is taken to indicate the type of storage (freshly allocated that should be `free`) and not as a literal call to `malloc` to allocate so many bytes.

As another example:

```
-sem( newstr, @p == (1p ? new[](1p) : 0) )
```

In words, this says that `newstr` is a function whose return value will, if the first argument is a non-null pointer, be the equivalent of a `new[]` of that size. Otherwise a NULL will be returned.

9.2.2.2 Return Semantic Validation

Return semantics are typically employed on functions for which the body is not available to PC-lint Plus but they can also be applied to functions that do have a visible definition. In this case, the return semantics will be validated against the actual function definition when analyzing specific calls. This can be used, for example, to document the return conditions that the function should employ and to have PC-lint Plus diagnose deviations from this contract.

Violation of return semantics are reported via Warning 2426. In the following example, the semantic `@p > 0` specifies that the pointer return value is never null. The implementation of this function contains a path that violates this semantic. PC-lint Plus will now report when such a path is taken causing the return value semantic to be violated:

```
//lint -sem(f, @p > 0) return value should never be null
void *f(int a, void *p) {
    if (a < 0)
        return 0;
    return p;
}
void g(void *p) {
    void *ptr = f(-1, p);
}
```

PC-lint Plus produces:

```
warning 2426: return value (nullptr) of call to function
    'f(int, void *)' conflicts with return semantic '(@p>0)'
    void *ptr = f(-1, p);
                ^
```

to indicate the violated on the semantic that specified the return value is never null.

When the return semantic conflicts with information collected during a specific call, the latter overrides the former. For example, despite the existence of the semantic claiming that the return value is not null, after the call to `f` PC-lint Plus will retain the knowledge gleaned from the actual call to `f` and diagnose an attempt to dereference the pointer. For example, if after the call to `f(-1, p)` we had:

```
int *iptr = ptr;
*ptr = 1;
    void
```

PC-lint Plus would issue:

```
warning 413: likely use of null pointer
    *iptr = 1;
    ^
supplemental 831: initialization yields nullptr
    int *iptr = ptr;
    ~~~~~^~~~~~~~~~
supplemental 831: initialization yields nullptr
    void *ptr = f(-1, p);
    ~~~~~~^~~~~~~~~~~~~~
supplemental 831: null to pointer conversion yields nullptr
        return 0;
               ^
```

If the `fso` flag is turned ON, return semantics will override any conflicting information obtaining during a specific walk although message **2426** will still be issued.

9.2.2.3 Function-wide semantics

An expression that is not a Return semantic is a 'Function-wide' semantic (to use the terminology of Section 9.1.1 Special Functions). It indicates a predicate that should be true. If there is a decided possibility that it is false, a diagnostic is issued.

What constitutes a "decided possibility"? This is determined by considerations described in Section 8 Value Tracking. If nothing is known about a situation, no diagnostic is issued. If what we do know suggests the possibility of a violation of the Function-wide semantic, a diagnostic is issued.

For example, to check to see if the region of storage passed to function `g()` is at least 6 bytes you may use the following:

```
//lint -sem( g, 1P >= 6 ) 1st arg. must have at least 6 bytes
void g(short *);
void f() {
    short a[3];            // a[] has 6 bytes
    short *p = a + 1;      // p points to 4 bytes
    g(a);   // OK
    g(p);   // Warning
}
```

Several constraints may be AND'ed using the `&&` operator. For example, to check that `fread(` ***`buffer`***`,` ***`size`***`,` ***`count`***`,` ***`stream`*** `)` has non-zero second and third arguments and that their product exactly equals the size of the buffer you may use the following option.

```
-sem( fread, 1P==2n*3n \&\& 2n>0 \&\& 3n>0 )
```

Note that we rely on C's operator precedence to properly group operator arguments.

To continue with our example we should add Return Semantics. `fread` returns a value no greater than the third argument (*count*). Also, the first and fourth arguments should be checked for null. A complete semantic option for `fread` becomes:

```
-sem( fread, 1P==2n*3n \&\& 2n>0 \&\& 3n>0, @n<=3n, 1p, 4p )
```

It is possible to employ macros in semantic expressions rather than hard numbers. For example:

```
//lint -sem( X::cpy, 1P <= BUFLEN )

char *strcpy(char *dest, const char *src);
#define BUFLEN 4

class X {
public:
    char buf[BUFLEN];
    void cpy(char *p) { strcpy(buf, p); }
    void slen(char *p);
};

void f(X &x) {
    x.cpy("abcd");  // Warning
    x.cpy("abc");   // OK
}
```

Just as is the case with `-function`, `-sem` may be applied to member functions. For example:

```
//lint -sem( X::cpy, 1P <= BUFLEN )

const int BUFLEN = 4;

class X
    {
  public:
    char buf[BUFLEN];
    void cpy( char * p )
        { strcpy( buf, p ); }
    void slen( char * p );
        };
    void f( X &x )
        {
        x.cpy( "abcd" );   // Warning
        x.cpy( "abc" );    // OK
        }
```

In this example, the argument to `X::cpy` must be less than or equal to `BUFLEN`. The byte requirements of `"abcd"` are 5 (including the nul character) and `BUFLEN` is defined to be 4. Hence a warning is issued here.

To specify semantics for template members, simply ignore the angle brackets in the name given to `-sem`. The semantics will apply to each template instantiation. For example, 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
    }
```

9.2.2.4 Overload-Specific Semantics

A user-defined semantic may be applied to a specific function overload by including the function's parameter list in the semantic specification (where a parameter list of `(void)` represents a function taking no arguments). For example:

```
//lint -sem(foo(int, int), chneg(1))
void foo(int);
void foo(int, int);

void bar() {
    foo(-8);          // Okay
    foo(-8, 20);      // Warning
}
```

A semantic can be applied to a specific function template instantiation by specifying the substituted template parameter types in the function parameter list:

```
//lint -sem(A1::rocker(int, char *, int), 3n <= 2P)
struct A1 {
    template <typename T2>
    int rocker(T2, char *, int);
};

void g() {
    char buf[10];
    A1 a1;
    a1.rocker(1, buf, 20);     // Warning
    a1.rocker(1.2, buf, 20);   // Okay
}
```

A semantic can be applied to all templated versions of a function by referencing the names of the template parameters in the argument list:

```
//lint -sem(A1::rocker(T2, char *, int), 3n <= 2P)
struct A1 {
    template <typename T2>
    int rocker(T2, char *, int);
};

void g() {
    char buf[10];
    A1 a1;
    a1.rocker(1, buf, 20);         // Warning
    a1.rocker(1.2, buf, 20);       // Warning
}
```

Every function call has 2 or 3 distinct monikers that can be used in a -sem option. Since the correct monikers might not be obvious in some scenarios, the monikers associated for each call will be provided via message

879 when the fsf flag is ON. If the fsf flag was enabled for the above example, the corresponding 879 messages would look like this:

```
info 879: semantic monikers are 'A1::rocker(int, char *, int)',
    'A1::rocker(T2, char *, int)', and 'A1::rocker'
    a1.rocker(1, buf, 20);
    ^


info 879: semantic monikers are 'A1::rocker(double, char *, int)',
    'A1::rocker(T2, char *, int)', and 'A1::rocker'
    a1.rocker(1.2, buf, 20);
    ^
```

The monikers are provided in order of decreasing specificity. The most specific moniker contains the complete parameter list. The next moniker contains the non-substituted template parameter names, this moniker does not exist for non-template functions. The most generic moniker is just the name of the function.

An overload set may have multiple semantics associated with it although only one semantic will be applied to a given function call, the semantic with the most specific matching function designator. For example:

```
//lint -sem(slow(T1, T1), 1n != 2n)
//lint -sem(slow(int, double), 1n > 0)
//lint -sem(slow, 1n > 1)

template <typename T1>
void slow(T1, T1);

void slow(int, double);
void slow(int);

void h() {
    slow(1, 0);     // Okay
    slow(0, 0);     // Warning, violates semantic for slow(T1, T1)

    slow(1, 3.0);   // Okay
    slow(0, 3.0);   // Warning, violates semantic for slow(int, double)

    slow(2);        // Okay
    slow(1);        // Warning, violates default semantic for slow
}
```

Note the difference between foo() and foo(void) in a semantic option. The former specifies that the semantic should apply to the C function named foo that does not have a prototype whereas the latter specifies a semantic for a function foo declared as taking no arguments (either by being declared as foo(void) in C or C++ or as foo() in C++).

9.2.3 Notes on Semantic Specifications

otes on Semantic Specifications

1. Every function has, potentially, a Return semantic (r), a Function-wide semantic (0), flag semantics (f), and Argument semantics for each of the arguments and the implied this argument (t). An expression of the form *i*p when it stands alone and is not part of another expression becomes an Argument semantic for argument *i* (presumably a pointer argument). Thus, for the option

   ```
   -sem( f, 2p, 1p > 0 )
   ```

   2p becomes an Argument semantic (the pointer should not be NULL) for argument 2. We can transfer this semantic to, say, the 3rd argument of function g by using the option

```
-function( f(2), g(3) )
```

The expression `1p>0` becomes the Function-wide semantic for function `f` and can be transferred via the `0` subscript as in:

```
-function( f(0), g(0) )
```

We could have placed these two together as one large semantic as in:

```
-sem( f, 2p \&\& 1p > 0 )
```

The earlier rendition is preferred because there is a specialized set of warning messages for the argument semantic of passing null pointers to functions.

2. Please note that `r_null` and an expression involving argument `@` are Return semantics. You cannot have both in one option. Thus you cannot have

   ```
   -sem( f, r\_null, @p = 1p )
   ```

   It is easy to convert this into an acceptable semantic as follows:

   ```
   -sem( f, @p == 0 || @p == 1p )
   ```

3. The notations for arguments and return values was not chosen capriciously. A notation such as `@n == 2n` may look strange at first but it was chosen so as not to conflict with user identifiers.

4. Please note that the types of arguments are signed integral values. Thus we may write

   ```
   -sem( strlen, @n < 1p )
   ```

   We are not comparing here integers with pointers. Rather we are comparing the number of items that a pointer points to (an integer) with an integral return value.

   For uniformity, the arithmetic of semantics is signed integral arithmetic, usually long precision. This means that greater-than comparisons with numbers higher than the largest signed long will not work.

10 MISRA Standards Checking

The Motor Industry Software Reliability Association (MISRA) is an organization that produces and maintains C and C++ programming guidelines. The primary purpose of these guidelines is to codify a set of recommendations related to software development that aids in the creation of "safe and reliable software". While MISRA is an effort born out of the automotive industry, MISRA's success has grown and the guidelines have been adopted to meet needs in other safety-critical industries such as healthcare and aerospace.

MISRA has produced three versions of their guidelines for C, each one replacing the previous version. The versions are MISRA C 1998 (sometimes referred to as MISRA C1), MISRA C 2004 (aka MISRA C2) and MISRA C 2012 (aka MISRA C3). In 2008, MISRA released guidelines for C++ (MISRA C++). While the MISRA C++ effort is currently defunct (there is no active work in this area), the guidelines are employed by some organizations seeking MISRA style guidelines for C++.

Each MISRA guidelines document consists of a series of numbered advisory, required, and mandatory "rules" and "directives". A directive is more generalized (such as requiring that "run-time failures be minimized") while Rules are concrete and testable (such as forbidding the use of C++ style comments). Directives are often not statically checkable while Rules often are.

PC-lint Plus provides support for the MISRA C2, MISRA C3, and MISRA C++ guidelines. This support is achieved through a combination of standard PC-lint Plus messages and elective notes dedicated to specific MISRA rules. Gimpel Software provides the author files `au-misra2.lnt`, `au-misra3.lnt`, and `au-misra-cpp.lnt` to enable the checks necessary to support these guidelines. These author files also include `-append` options, which cause messages that are used to report on MISRA violations to be annotated with the corresponding Rule or Directive number(s).

While some of the messages are very specific to MISRA guidelines (such as those involving interactions amongst "essential types", a MISRA creation), any of the messages may be employed individually for those desiring to make use of a subset of the checks, outside of MISRA compliance checking.

The author files enable checks for both library and non-library code. This means that the standard headers employed by your source code are subject to the same scrutiny as the rest of the project. This is often a requirement but can result in a lot of noise if library code is not subject to the same compliance requirements as the rest of the project. The simplest way to disable MISRA checks for library code is to place the options `-wlib(4) -wlib(1)` immediately after the author file is referenced. This raises and immediately lowers the warning level for libraries resulting in a suppression of all non-error messages from library code. Any non-error messages that you intend to enable for library code (e.g. via `+elib`) should appear after these options.

The following subsections document the level of support provided by PC-lint Plus for each of the directives and rules supported. For each rule, the rule number, headline text, and primary enforcing messages are provided. The letter in parenthesis after each rule indicates whether the rule is advisory (A), required (R), or mandatory (M). An asterisk beside this letter indicates that MISRA has deemed the rule to be "undecidable", that is not possible to be fully checked by static analysis methods. In such cases PC-lint Plus provides the level of support feasible. While every effort is made to ensure the correctness of the information provided here, the author files should be considered the ultimate source of enforcement information.

10.1 MISRA C 2012

10.1.1 Supported MISRA C 2012 Directives

Directive	Message
4.4 (A*)	602
4.5 (A*)	9046
4.6 (A*)	970
4.7 (R*)	534
4.8 (A*)	9045
4.9 (A*)	9026
4.10 (R*)	451
4.11 (R*)	418 419 420 422 668 669 670 671 2423 2623
4.12 (R*)	586

10.1.2 Supported MISRA C 2012 Rules

Rule	Message
1.3 (R*)	2454 9020 9023
2.1(R*)	506 527 681 827
2.2 (R*)	438 505 520 521 522
2.3 (A)	751 756
2.4 (A)	753 9058
2.5 (A)	750 755
2.6 (A)	563
2.7 (A)	715
3.1 (R)	602 9059 9066 9259
4.1 (R)	9039
4.2 (A)	584 739 9060
5.1 (R)	621
5.2 (R)	621
5.3 (R)	578 621
5.4 (R)	547 760 621
5.5 (R)	9095 9096
6.1 (R)	9149
6.2 (R)	9088
7.1 (R)	9001
7.2 (R)	9048
7.3 (R)	620 9057
7.4 (R)	489 1776 1778
8.1 (R)	601 808
8.2 (R)	936 937 955
8.3 (R)	9072 9073 9094
8.4 (R)	957 9075
8.5 (R)	9004
8.7 (A)	765
8.8 (R)	839
8.9 (A)	9003
8.11 (A)	9067

Rule	Message
8.12 (R)	488
8.13 (A*)	818 844 954
8.14 (R)	586
9.1 (M*)	530 644
9.2 (R)	9069
9.3 (R)	9068
9.4 (R)	485
9.5 (R)	9054
10.1 (R)	9027
10.2 (R)	9028
10.3 (R)	9034
10.4 (R)	9029
10.5 (A)	9030
10.6 (R)	9031
10.7 (R)	9032
10.8 (R)	9033
11.1 (R)	9074
11.2 (R*)	9076
11.3 (R)	9087
11.4 (A)	9078
11.5 (A)	9079
11.7 (R)	177 179 9295
11.8 (R)	9005
11.9 (R)	9080
12.1 (A)	9050
12.2 (R*)	9053
12.3 (A)	9008
12.4 (A)	648
13.1 (R*)	446
13.2 (R*)	564 931
13.3 (A)	9049
13.4 (A)	720 820 9084
13.5 (R*)	9007
13.6 (M)	9006 9089
14.1 (R*)	9009
14.3 (R)	650 685 774
14.4 (R)	9036
15.1 (A)	801
15.2 (R)	9064
15.3 (R)	9041
15.4 (A)	9011
15.5 (A)	904
15.6 (R)	9012
15.7 (R)	9013 9063
16.1 (R)	9014 9042 9077 9081 9082 9085
16.2 (R)	9055
16.3 (R)	9077 9090
16.4 (R)	9014 9085

Rule	Message
16.5 (R)	9082
16.6 (R)	9081
16.7 (R)	483
17.1 (R)	829
17.2 (R*)	9070
17.3 (M)	718
17.4 (M)	533
17.5 (A*)	473
17.6 (M)	9043
17.7 (R)	534
17.8 (A*)	9044
18.1 (R*)	415 416 428 661 662 676
18.2 (R*)	947
18.3 (R*)	946
18.4 (A)	9016
18.5 (A)	9025
18.6 (R*)	733 789 604
18.7 (R)	9038
18.8 (R)	9035
19.2 (A)	9018
20.1 (A)	9019
20.2 (R)	9020
20.3 (R)	12 544
20.4 (R)	9051
20.5 (A)	9021
20.6 (R)	436
20.7 (R)	665
20.8 (R)	9037
20.9 (R)	553
20.10 (A)	9024
20.11 (R)	484
20.12 (R)	9015
20.13 (R)	16 544 9160
20.14 (R)	8
21.1 (R)	9071 9083
21.2 (R)	683
21.3 (R)	586
21.4 (R)	586 829
21.5 (R)	586 829
21.6 (R)	586
21.7 (R)	586
21.8 (R)	586
21.9 (R)	586
21.10 (R)	586 829
21.11 (R)	829
21.12 (A)	586
22.1 (R*)	429
22.2 (M*)	424 449

Rule	Message
22.5 (M*)	9047

10.1.3 Supported MISRA C 2012 AMD-1 Rules

Directive	Message
12.5 (M)	682 882
21.13 (M)	426
21.15 (R)	857
21.16 (R)	9098
21.17 (M*)	419 420

10.2 MISRA C++

10.2.1 Supported MISRA C++ Rules

Rule	Message
0-1-1 (R)	506 527 681 685 774 827 944
0-1-2 (R)	685 774 827 944
0-1-3 (R)	528 529 714 752 757 1715
0-1-4 (R)	528 529 550 551 552
0-1-5 (R)	751 753 756 758
0-1-6 (R)	438 838
0-1-7 (R)	534
0-1-8 (R)	9175
0-1-9 (R)	438 587 685 774 838 944 948
0-1-10 (R)	528 714 1714 1914
0-1-11 (R)	715
0-1-12 (R)	715
0-3-2 (R)	534
2-3-1 (R)	584 739
2-5-1 (A)	9102
2-7-1 (R)	602
2-10-1 (R)	9046
2-10-2 (R)	578 1411 1511 1516
2-10-5 (A)	9103
2-10-6 (R)	18
2-13-1 (R)	606
2-13-2 (R)	9104
2-13-3 (R)	9105
2-13-4 (R)	9106
2-13-5 (R)	707
3-1-1 (R)	9107
3-1-2 (R)	9108
3-1-3 (R)	9067
3-2-1 (R)	18 31
3-2-2 (R)	15 31
3-2-3 (R)	9004
3-2-4 (R)	15 31
3-3-1 (R)	759 765
3-3-2 (R)	401 839
3-9-1 (R)	9073 9094 9168
3-9-2 (A)	970
3-9-3 (R)	9110
4-5-1 (R)	9111
4-5-3 (R)	9112
4-10-2 (R)	910
5-0-1 (R)	564
5-0-2 (A)	9113
5-0-3 (R)	9114 9116
5-0-4 (R)	9117

Rule	Message
5-0-5 (R)	9115 9118
5-0-6 (R)	9119 9120
5-0-7 (R)	9121 9122
5-0-8 (R)	9123 9124
5-0-9 (R)	9125
5-0-10 (R)	9126
5-0-11 (R)	9128
5-0-13 (R)	909
5-0-14 (R)	909
5-0-15 (R)	947 9016
5-0-16 (R)	415 416 661 662
5-0-17 (R)	947
5-0-18 (R)	946
5-0-19 (R)	9025
5-0-20 (R)	9172
5-0-21 (R)	9130
5-2-1 (R)	9131
5-2-2 (R)	1774 1939
5-2-3 (A)	9171
5-2-4 (R)	1924
5-2-5 (R)	9005
5-2-6 (R)	611
5-2-7 (R)	916 920 923 926 927 928 929 930 9176
5-2-8 (A)	923 925 930
5-2-9 (A)	9091
5-2-10 (A)	9049
5-2-11 (R)	1753
5-2-12 (R)	9132
5-3-1 (R)	9133
5-3-2 (R)	9134
5-3-3 (R)	9135
5-3-4 (R)	9006
5-8-1 (R)	9136
5-14-1 (R)	9007
5-18-1 (R)	9008
5-19-1 (A)	648
6-2-1 (R)	720 820 9084
6-2-2 (R)	9137
6-2-3 (R)	9138
6-3-1 (R)	9012
6-4-1 (R)	9012
6-4-2 (R)	9013
6-4-3 (R)	9042
6-4-4 (R)	9055
6-4-5 (R)	9090
6-4-6 (R)	744 9139
6-4-7 (R)	483
6-4-8 (R)	764

Rule	Message
6-5-3 (R)	850
6-6-1 (R)	9041
6-6-2 (R)	107 9064
6-6-3 (R)	9254
6-6-4 (R)	9011
6-6-5 (R)	904
7-1-1 (R)	952
7-1-2 (R)	818
7-3-1 (R)	9141 9162
7-3-2 (R)	9142
7-3-3 (R)	1751
7-3-4 (R)	9144
7-3-6 (R)	9145
7-4-2 (R)	
7-5-1 (R)	604
7-5-2 (R)	789
7-5-3 (R)	1780 1940
7-5-4 (A)	9070
8-0-1 (R)	9146
8-3-1 (R)	1735
8-4-1 (R)	9165
8-4-2 (R)	9072 9272
8-4-3 (R)	533
8-4-4 (R)	9147
8-5-1 (R)	530
8-5-2 (R)	940
8-5-3 (R)	9148
9-3-1 (R)	605 1536
9-3-2 (R)	1536
9-3-3 (R)	1762
9-5-1 (R)	9018
9-6-2 (R)	9149
9-6-3 (R)	9149
9-6-4 (R)	9088
10-1-1 (A)	9174
10-1-3 (R)	1748
10-3-2 (R)	1909
10-3-3 (R)	9170
11-0-1 (R)	9150
12-1-1 (R)	1506
12-1-2 (A)	1928
12-1-3 (R)	9169
12-8-1 (R)	1938
12-8-2 (R)	9151
14-5-2 (R)	1789
14-5-3 (R)	1721
14-7-1 (R)	1795
14-7-3 (R)	1576 1577

Rule	Message
14-8-2 (A)	9153
15-0-2 (A)	9154
15-0-3 (R)	646
15-1-2 (R)	1419
15-1-3 (R)	9156
15-3-1 (R)	1546
15-3-4 (R)	1560
15-3-5 (R)	1752
15-3-7 (R)	1127
15-4-1 (R)	1548
15-5-1 (R)	1546
15-5-2 (R)	1549
15-5-3 (R)	1546
16-0-1 (R)	9019
16-0-2 (R)	9158 9159
16-0-3 (R)	9021
16-0-4 (R)	9026
16-0-5 (R)	436
16-0-6 (R)	9022
16-0-7 (R)	553
16-0-8 (R)	16 544 9160
16-1-1 (R)	491
16-1-2 (R)	8
16-2-3 (R)	967
16-2-4 (R)	9020
16-2-5 (A)	9020
16-2-6 (R)	12
16-3-1 (R)	9023
16-3-2 (A)	9024
17-0-1 (R)	9052 9071
17-0-2 (R)	9093
17-0-5 (R)	586
18-0-1 (R)	829
18-0-2 (R)	586
18-0-3 (R)	586
18-0-4 (R)	829
18-0-5 (R)	586
18-2-1 (R)	586
18-4-1 (R)	586 9173
18-7-1 (R)	829
19-3-1 (R)	586
27-0-1 (R)	829

10.3 MISRA C 2004

10.3.1 Supported MISRA C 2004 Rules

Rule	Message #
1.2 (R)	many
2.1 (R)	586
2.2 (R)	9260
2.3 (R)	602
2.4 (A)	602
3.4 (R)	975
4.1 (R)	606 2406 9104 9204
4.2 (R)	584 739
5.2 (R)	578
6.1 (R)	9128 9209
6.2 (R)	9128
6.3 (A)	970
6.4 (R)	9212
6.5 (R)	9088 9288
7.1 (R)	9001 9104
8.1 (R)	718 746 937 957
8.2 (R)	601 808
8.3 (R)	9073 9094
8.4 (R)	15 18 64
8.5 (R)	9107
8.6 (R)	9108
8.7 (R)	9003
8.8 (R)	9004
8.10 (R)	765
8.11 (R)	401 839
8.12 (R)	9067
9.1 (R)	530 644
9.2 (R)	576 940 9068
9.3 (R)	9148
10.1 (R)	9225 9226
10.2 (R)	9227 9228
10.3 (R)	9229
10.4 (R)	9230
10.5 (R)	9231
10.6 (R)	9048
11.1 (R)	176 178 9237
11.2 (R)	177 179
11.3 (A)	923
11.4 (A)	9087 9287
11.5 (R)	9005
12.1 (A)	9050
12.2 (R)	564
12.3 (R)	9006 9089
12.4 (R)	9007

Rule	Message #
12.5 (R)	9240
12.6 (A)	9232
12.7 (R)	9233
12.8 (R)	9234
12.9 (R)	9235
12.10 (R)	9008
12.11 (A)	648
12.12 (R)	9110
12.13 (A)	9049
13.1 (R)	720 9236
13.2 (A)	9224
13.3 (R)	777 9252
13.4 (R)	9009
13.5 (R)	440 443
13.6 (R)	850
13.7 (R)	506, 650 685 774 845
14.1 (R)	527 681 827
14.2 (R)	505 522
14.3 (R)	9138
14.4 (R)	801
14.5 (R)	9254
14.6 (R)	9011
14.7 (R)	904
14.8 (R)	9012
14.9 (R)	9012
14.10 (R)	9013 9063
15.0 (R)	9042
15.1 (R)	44 9055
15.2 (R)	9090
15.3 (R)	9014 9139
15.4 (R)	9238
15.5 (R)	764
16.1 (R)	9165
16.2 (R)	9070
16.3 (R)	955
16.4 (R)	9072
16.5 (R)	937
16.6 (R)	118 119
16.7 (A)	818
16.8 (R)	533
16.9 (R)	9147
16.10 (R)	534
17.4 (R)	9016, 9017 9264
17.5 (A)	9025
17.6 (R)	604 733 789
18.1 (R)	115
18.4 (R)	9018
19.1 (A)	9019

Rule	Message #
19.2 (A)	9020
19.3 (R)	12
19.5 (R)	9158 9159
19.6 (R)	9021
19.7 (A)	9026
19.8 (R)	131
19.9 (R)	436
19.10 (R)	9022
19.11 (R)	553
19.12 (R)	9023
19.13 (A)	9024
19.14 (R)	491
19.15 (R)	451
19.16 (R)	16 544 9160
19.17 (R)	8
20.1 (R)	980 9071 9083
20.2 (R)	9093
20.4 (R)	586
20.5 (R)	586
20.6 (R)	586
20.7 (R)	586
20.8 (R)	586 829
20.9 (R)	829
20.10 (R)	586
20.11 (R)	586
20.12 (R)	586

11 Other Features

11.1 Format Checking

The `printf`-like and `scanf`-like functions are fertile ground for programming errors as they are not type safe due to their variadic nature, and incorrect use often involves undefined behavior that is not diagnosed by compilers. PC-lint Plus performs comprehensive analysis of the use of these functions diagnosing format incompatibilities, inconsistent and redundant specifier combinations, missing and unused arguments, mis-use of positional specifiers, use of non-standard conversion specifiers, unbounded conversions, and other anomalies.

There are several categories of checking performed and over two dozen messages dedicated to analysis of format string functions.

11.1.1 Dangerous Use

The messages in this section focus on particularly egregious errors that always have the potential to result in undefined behavior. A relatively common error is to provide fewer data arguments than required by the format string. This often happens for particularly large format strings or when the format string is changed. Another possibility is a missing comma between string literal arguments such as in:

```
printf("\%10s \%s", "Name" "Value");
```

which will be diagnosed with:

```
warning 558: too few data arguments for format string (1 missing)
    printf("%10s %s", "Name" "Value");
                  ~^
```

showing the location of the first conversion specifier without a value and the total number of missing data arguments. Such a call results in undefined behavior as `printf` processes data on the stack looking for the next argument.

The `scanf`-like functions have additional potential concerns. Using the `%s` or `%[` conversion specifier without a maximum field width will result in undefined behavior if the stored string exceeds the provided buffer, e.g.:

```
char buf[10];
scanf("%s", buf);
```

which will result in:

```
warning 498: unbounded scanf conversion specifier 's' may result in buffer overflow
    scanf("%s", buf);
           ~^
```

This message will not be issued if the non-standard 'm' prefix is used (e.g. `%ms`), which specifies that `scanf` should dynamically allocate a buffer large enough to hold the result.

A missing closing bracket for the `%[` conversion specifier is yet another instance of undefined behavior and is diagnosed with message 2406. For example:

```
char buf[100];
scanf("%99[^]", buf);
```

Here the programmer intended to store a series of consecutive ^ characters into `buf` but a special exception for the `scanf` function causes the closing bracket to be considered part of the pattern in this case, not the closing bracket to the `%[` conversion specifier. PC-lint Plus will issue:

```
warning 2406: no closing ']' for '%[' in scanf format string
    scanf("%99[^]", buf);
           ~~^
```

Another source of undefined behavior stemming from `scanf` is when a field width of zero is specified:

```
scanf("\%0s", buffer);
```

which will elicit:

```
warning 2407: zero field width in scanf format string is unused
    scanf("%0s", buffer);
            ^
```

Less common is a format string that is not null terminated, this can happen when a sized array of `char` is initialized in a way that prevents the terminating NUL character from being appended, e.g.:

```
const char fmt[2] = "\%s";
```

which when used as a format string will result in warning 496 (the declaration alone is enough to prompt info 784).

Finally, passing a wide format string to a non-wide format function will be diagnosed with warning 2409.

11.1.2 Argument Inconsistencies

This group of messages will diagnose arguments to format functions that do not match the corresponding conversion specifier in the format string. Warning 559 is issued for significant discrepancies:

```
printf("\%s", 12);
```

will elicit:

```
warning 559: format '%s' specifies type 'char *' which is inconsistent with
argument no. 2 of type 'int'
    printf("%s", 123);
            ~~   ^~~
```

Messages 705 and 706 are used to diagnose "nominal" inconsistencies between the expected and actual type or the type pointed to. A "nominal" difference means that the size and basic type of the argument was correct but the type was not exactly the type prescribed by the Standard, e.g. a difference in sign. `printf`-like functions can accept a star (`*`) in place of the field width and/or precision in which case the value is taken from the next `int` argument. If this argument is missing warning 2402 is issued. Warning 2403 is issued if the type of this argument is incorrect.

11.1.3 Positional Arguments

The POSIX positional argument syntax allows arguments to be referred to by number using the syntax *n*$ where *n* refers to a data argument. For example the following two calls to `printf` are equivalent:

```
printf("%d %d", 1, 2);      // ISO syntax
printf("%1$d %2$d", 1, 2);  // POSIX positional notation
```

Positional arguments allow format strings to reference arguments in an order that is different from how they are supplied to the format function as well as using the same argument multiple times in the format string. There are several caveats when using positional arguments. For starters, the position starts at 1, not 0; an attempt to use 0 as a position will elicit message 493. Positional arguments cannot be mixed with non-positional arguments in the same format string, violations of this rule are diagnosed by message 2401. Referencing a non-existent positional argument will be diagnosed by messages 494 (data argument positions) and 2404 (field width and field precision positions).

11.1.4 Non-ISO features

Features that are not specified by the ISO C Standard may not be portable to other platforms and their use can be diagnosed by PC-lint Plus. These features include positional arguments described above (message 855), non-ISO format specifiers such as `%m` for `printf`-like functions and `%C`, and `%D` for `printf`-like and `scanf`-like functions (message 816), and non-standard length modifiers / conversion specifier combinations (message 499). Since the behavior of these features are not specified by the Standard, their use on platforms that do not support them may result in unintended or undefined behavior.

11.1.5 Incorrect Format Specifiers

There are several reasons that a conversion specifier may be invalid and PC-lint Plus will diagnose these.

1. An incomplete format specifier (e.g. `%h`) will be diagnosed by warning 492,
2. an unknown conversion specifier (e.g. `%b`) by warning 557,
3. inconsistent or redundant format specifiers (e.g. `%+u`) are diagnosed by warning 566, and
4. illegal use of a field width or precision with a conversion specifier will result in warning 2405.

Each of these messages represent a programming error or the use of extensions that PC-lint Plus is not aware of. For example:

```
printf("\%.10c", 'a');
```

will result in:

```
warning 2405: precision used with 'c' conversion specifier is undefined
    printf("%.10c", 'a');
            ~^~~~
```

A precision is not allowed with the `%c` conversion specifier and providing one results in undefined behavior.

11.1.6 Suspicious Format Specifiers

There are several suspicious constructs that by themselves do not represent errors but are sufficiently unusual to warrant review. This includes

1. an empty format string (message 497),
2. a format string that contains an embedded NUL character (message 495),
3. the use of a non-literal format string (messages 592 and 905),
4. unused data arguments (message 719).

11.1.7 Elective Notes and Customization

Message 983 will point out uses of dash(`-`) within a `scanf` scan-list (e.g. `%[A-Z]`). As the behavior of the dash in this position is implementation defined, some implementations interpret this as a range, others do not.

`printf` and `scanf` conversion specifiers can also be deprecated using the `-deprecate` option, which will cause message 586 to be emitted when they are seen. E.g. `-deprecate(printf_code, n)` will cause a warning to be issued whenever the `%n` conversion specifier is used in a `printf`-like function. Similarly, use `scanf_code` to deprecate `scanf` conversion specifiers. See `-deprecate` for additional information.

The `-printf` and `-scanf` options allow a user to specify functions that resemble a member of the `printf` or `scanf` family. PC-lint Plus has built-in support for the following formatting functions, including those from Annex K in the C11 Standard:

printf-like functions	scanf-like functions	Annex K printf-like functions	Annex K scanf-like functions
fprintf	fscanf	fprintf_s	fscanf_s
fwprintf	fwscanf	fwprintf_s	fwscanf_s
printf	scanf	printf_s	scanf_s
snprintf	sscanf	snprintf_s	sscanf_s
sprintf	swscanf	snwprintf_s	swscanf_s
swprintf	vfscanf	sprintf_s	vfscanf_s
vfprintf	vscanf	swprintf_s	vfwscanf_s
vprintf	vsscanf	vfprintf_s	vscanf_s
vsnprintf	wscanf	vfwprintf_s	vsscanf_s
vsprintf		vprintf_s	vswscanf_s
wprintf		vsnprintf_s	vwscanf_s
		vsnwprintf_s	wscanf_s
		vsprintf_s	
		vswprintf_s	
		vwprintf_s	
		wprintf_s	

11.2 Precision, Viable Bit Patterns, and Representable Values

Several messages (including 650, 587, and 734) deal with the notion of "precision" or otherwise involve static dermination of whether or not a value is representable in a particular context. In PC-lint Plus precision has been expanded from covering only the conceptual width of a value (e.g. a bitfield or right-shifted variable) to encompass the potential bit patterns that can result from the use of bitwise operators, addition, subtraction, or values of `enum` type. Many such messages utilize supplemental messages to convey bit pattern information when relevant. For example, for some `unsigned int u`:

```
if ( (u & 0x10) == 0x11 ) { }
```

will result in:

```
warning 587: predicate '==' can be pre-determined
    and always evaluates to false
    if( (u & 0x10) == 0x11 ) { }
        ~~~~~~~~~~ ^  ~~~~
supplemental 891: incompatible bit patterns:
    U32_00000000000000000000000000010001 vs
    U32_000000000000000000000000000?0000
    if ( (u & 0x10) == 0x11 ) { }
                    ^
```

where `U32_00000000000000000000000000010001` is the binary representation of the constant `0x11` (`17`) as an unsigned 32-bit integer and `U32_000000000000000000000000000?0000` shows the potential values of the left operand which may or may not have the bit in the sixteens place set but cannot have the bit in the ones place set.

In a more complex example, the effects of implicit conversions may be visible, for example:

```
void f(char c1, char c2) {
    if ( (c1 & 13) + (c2 & 8) == 6 ) { }
}
```

will result in:

```
warning 587: predicate '==' can be pre-determined and always evaluates to false
    if ( (c1 & 13) + (c2 & 8) == 6 ) { }
         ~~~~~~~~~~~~~~~~~~~~ ^  ~
```

```
supplemental 891: incompatible bit patterns:
    S32_00000000000000000000000000000110 vs
    S32_............................??0?
    if ( (c1 & 13) + (c2 & 8) == 6 ) { }
                              ^
```

The first bit pattern represents the constant 6. The second bit pattern is masked with periods beyond its meaningful precision because it is signed in order to reduce confusion regarding the potential sign extension of an inexact value. This message is indicating that the resultant sum cannot ever have the bit in the twos place set and therefore cannot represent the value to which it is being compared.

Supplemental messages displaying bit patterns will only appear when they will provide useful information beyond that conveyed by the precision specified in the original message. For example:

```
void f(unsigned char uc) {
    if (uc == -1) { }
}
```

will result in:

```
warning 650: constant '-1' out of range for operator '=='
    if (uc == -1) { }
           ^
```

with no accompanying supplemental message.

The following message numbers currently utilize the unified precision and viable bit pattern architecture:

#	Context	Category
572	>> or >>= by a constant	loss of precision
587	==, !=, <, <=, >, or >= with one constant operand	pre-determined predicate
650	==, !=, <, <=, >, or >= with one constant operand, switch case	pre-determined predicate
685	<, <=, >, or >= with one constant operand	pre-determined predicate
734	assignment	loss of precision

11.3 Static Initialization

Traditional lint Compilers do not flag uninitialized static (or global) variables because the C/C++ language defines them to be 0 if no explicit initialization is given. But uninitialized statics, because they can cover such a large scope, can be easily overlooked and can be a serious source of error. Additionally, some embedded compilers do not perform this standard mandated implicit initialization. PC-lint Plus will flag static variables (see messages 727, 728 and 729) that have no initializer and that are assigned no value. For example, consider:

```
int n;
int m = 0;
```

There is no real difference between the declarations as far as C/C++ is concerned but PC-lint Plus regards `m` as being explicitly initialized and `n` not explicitly initialized. If `n` is accessed by nowhere assigned a value, a complaint will be emitted.

11.4 Indentation Checking

Indentation checking can be used to locate the origins of missing left and right braces. It can also locate potential problems in a syntactically correct program. For example, consider the code fragment:

```
if( ... )
    if( ... )
        statement
else statement
```

Apparently the programmer thought that the `else` associates with the first `if` whereas a compiler will, without complaint, associate the `else` with the second `if`. PC-lint Plus will signal that the `else` is negatively indented with respect to the second `if`.

There are three forms of messages; Informational 725 is issued in the case where there is no indentation (no positive indentation) when indentation is expected, Warning 525 is issued when a construct is indented less than (negatively indented from) a controlling clause, and 539 is issued when a statement that is not controlled by a controlling clause is nonetheless indented from it.

Of importance in indentation checking is the weight given to leading tabs in the input file. Leading tabs are by default regarded as 8 blanks but this can be overridden by the `-t#` option. For example `-t4` signifies that a tab is worth 4 blanks (see the `-t#` option in Section 4.3.3 Message Presentation).

Recognizing indentation aberrations comes dangerously close to advocating a particular indentation scheme; this we wish to avoid. For example, there are at least three main strategies for indentation illustrated by the following templates:

```
if( e ) {
    statements
}

if( e )
    {
    statements
    }

if( e )
{
    statements
}
```

Whereas the indentation methods appear to differ radically, the only real difference is in the way braces are handled. Statements are always indented positively from the controlling clause. For this reason PC-lint Plus makes what is called a *strong* check on statements requiring that they be indented (or else a 725 is issued) and only a *weak* check on braces requiring merely that they not be negatively indented (or else a 525 is issued).

`case`, and `default` undergo a weak check. This means, for example, that

```
switch()
    {
case 'a' :
    break;
default:
break;
    }
```

raises only the informational message (725) on the second `break` but no message appears with the `case` and `default` labels.

The `while` clause of a `do ... while(e);` compound undergoes a weak check with respect to the `do`, and an `else` clause undergoes a weak check with respect to its corresponding `if`.

An `else if()` construct on the same line establishes an indentation level equal to the location of the `else` not the `if`. This permits use of the form:

```
if()
    statement}
```

```
else if()
    statement
else if()
    statement
...
else
    statement
```

Only statement beginnings are checked. Thus a comment can appear anywhere on a line and it will not be flagged. Also a long string (if it does not actually begin a statement) may appear anywhere on the line.

A label may appear anywhere unless the `+fil` flag is given (Section 4.10 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.

11.5 Size of Scalars

Since the user of PC-lint Plus has the ability to set the sizes of various data objects (See the `-s`.. options in Section 4.5.1 Scalar Data Size), the reader may wonder what the effect would be of using various sizes.

Several of the loss of precision messages (`712`, `734`, `735` and `736`) depend on a knowledge of scalar sizes. The legitimacy of bit field sizes depends on the size of an `int`. Warnings of format irregularities are based in part on the sizes of the items passed as arguments.

One of the more important effects of type sizes is the determination of the type of an expression. The types of integral constants depend upon the size of `int` and `long` in ways that may not be obvious. For example, even where `int` are represented in 16 bits the quantity:

```
35000
```

is `long` and hence occupies 4 (8-bit) bytes whereas if `int` is 32 bits the quantity is a four byte `int`. If you want it to be `unsigned` use the `u` suffix as in `35000u` or use a cast.

Here are the rules: the type of a decimal constant is the first type in the list (`int`, `long`, `long long`) that can represent the value. The maximum values for these types are taken to be $2^{\texttt{sizeof(}\mathit{type}\texttt{)}*\mathit{bits\text{-}per\text{-}byte\text{-}1}} - 1$. The quantities `sizeof(int)`, `sizeof(long)`, and `sizeof(long long)` are based on the `-si#`, `-sl#`, and `-sll#` options respectively. The type of a hex or octal constant, however, is the first type on the list (`int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`).

For any constant (decimal, hex or octal) with a `u` or `U` suffix, one selects from the list (`unsigned int`, `unsigned long`, `unsigned long long`). If an `l` or `L` suffix, the list is (`long`, `long long`) for decimal constants and (`long`, `unsigned long`, `long long`, `unsigned long long`) for hex and octal constants. If both suffixes are used (e.g. `UL`), the list is (`unsigned long`, `unsigned long long`) for any constant. If the suffix is `ll` or `LL`, the type is `unsigned long long` for decimal constants and either `long long` or `unsigned long long` for hex and octal constants. Finally, constants containing both the `u`/`U` and `ll`/`LL` suffixes are always of type `unsigned long long`, regardless of base.

The size of scalars enters into the typing of intermediate expressions in a computation. Following ANSI/ISO standards, PC-lint Plus uses the so-called *value-preserving* rule for promoting types. Types are promoted when a binary operator is presented with two unlike types and when unprototyped function definitions specify subinteger parameters. For example, if an `int` is added to an `unsigned short`, then the latter is converted to `int` provided that an `int` can hold all values of an `unsigned short`; otherwise, they are both converted to `unsigned int`. Thus the signedness of an expression can depend on the size of the basic data objects.

11.6 Stack Usage Report

```
+stack(sub-option,...)
-stack(sub-option,...)
```

The `+stack` version of this option can be used to trigger a stack usage report. The `-stack` version is used only to establish a set of options to be employed should a `+stack` option be given. To prevent surprises if a `-stack` option is given without arguments it is taken as equivalent to a `+stack` option.

The sub-options are:

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

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

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

`&external(`*n*`)` establishes an assumption that each external function (that is not given an explicit stack requirement, see below) requires *n* bytes of stack. By default this value is 32.

`&summary` This option indicates that the programmer is interested in at least a summary of stack usage (stack used by the worst case function). The summary comes in the form of Elective Note 974 and is equivalent to issuing the option `+e974`. This option is not particularly useful since a summary report will automatically be given if a `+stack` option is given. It is provided for completeness.

name`(`*n*`)` where *name* is the name of a function, explicitly designates the named function as requiring *n* bytes of total stack. This is typically used to provide stack usage values for functions whose stack usage could not be computed either because the function is involved in recursion or in calls through a function pointer. *name* may be a qualified name.

Example:

```
+stack( \&file=s.txt, alpha(12), A::get(30) )
```

requests a stack report to be written to file `s.txt` and further, that function `alpha()` requires 12 bytes of stack and function `A::get()` requires 30.

At global wrap-up, a record is written to the file for each defined function. The records appear alphabetized by function name.

Each record will contain the name of a function followed by the amount of auto storage required by its local auto variables. Note that auto variables that appear in different and non-telescoping blocks may share storage so the amount reported is not simply the sum of the storage requirements of all auto variables.

Each function is placed into one of seven categories as follows:

1. *recursive loop* – a function is *recursive loop* if it is recursive and we can provide a call to a function such that that call is in a recursive loop that terminates with the original function. Thus the function is not

merely recursive but demonstrably recursive. The record contains the name of a function called and it is guaranteed that the called function will also be reported as *recursive loop*.

It is assumed that any recursive function requires an unbounded amount of stack. If that assumption is incorrect and you can deduce an upper bound of stack usage, then you can employ the `+stack` option to indicate this upper bound. In a series of such moves you can convert a set of functions containing recursion to a set of functions with a known bound on the stack requirements of each function.

2. *recursive* – a function is designated as *recursive* if it is recursive but we do not provide a specific circular sequence of calls to demonstrate the fact. Thus the function is recursive but unlike *recursive loop* functions it is not demonstrably recursive. The record contains the name of a function called. This function will either be *recursive loop*, *recursive* or *calls recursive* (see next category). If you follow the chain of calls it is guaranteed that you will ultimately arrive at a function that is labeled *recursive loop*.

3. *calls recursive* – a function may itself be non-recursive but may call a function (directly or indirectly) that is recursive. The stack requirements of functions in this category are considered to be unbounded. The record will contain the name of a function that it calls. This function will either be '*recursive loop*', '*recursive*' or '*calls recursive*'. If you follow the chain of calls it is guaranteed that you will ultimately arrive at a function that is labeled '*recursive loop*'.

4. *non-deterministic* – a function is said to be *non-deterministic* if it calls through a function pointer. The presumption is that we cannot determine by static means the set of functions so called. No function is labeled *non-deterministic* unless it is first determined that it is not in the *recursive* categories. That is, it could not be determined following only deterministic calls that it could reach a *recursive* function.

   If you can determine an upper bound for the stack requirements of a non-deterministic function then, like a recursive function, you may employ the `+stack` option to specify this bound and in a sequence of such options determine an upper bound on the amount of stack required by the application.

5. *calls a non-deterministic function* – a function is placed into this category if it calls directly or indirectly a *non-deterministic function*. It is guaranteed that we could not find a recursive loop involving this function or even a deterministic path to a recursive function. The record will be accompanied by the name of a function called. It is guaranteed that if you follow the chain of calls you will reach a non-deterministic function.

6. *finite* – a function is *finite* if all call chains emanating from the function are bounded and deterministic. The record will contain a total stack requirement. This will be a worst case stack usage. The record will bear the name of a function called (or 'no function' if it does not call a function). If you follow this chain you will pass through a (possibly zero length) sequence of finite functions before arriving at a function that

   (a) is labeled as '*finite*' but calls no other function or
   (b) is labeled as '*external*' or
   (c) is labeled as '*explicit*' (see next category).

   You should be able to confirm the stack requirements by adding up the contribution from each function in the chain plus a fixed call overhead for each call. The amount of call overhead can be controlled by the `stack` option.

   For '*external*' functions there is an assumed default stack requirement. You may employ the `+stack` option to specify the stack requirement for a specific function or to alter the default requirement for external functions.

7. *explicit* – a function is labeled as *explicit* if there was an option provided to the `-stack` option as to the stack requirements for a specific function.

Stack Report Formatting Options

The information provided by this option can be formatted by the user using the `-format_stack` option. This allows the information to be formatted to a form that would allow it to be used as input to a database or to a spreadsheet. This format can contain escape codes

'`%f`' for the function name
'`%a`' for the local auto storage
'`%t`' for type (i.e. one of the seven categories above)
'`%n`' for the total stack requirement
'`%c`' for the callee and
'`%e`' for an 'external' tag on the callee

See `-format_stack` for more details. See also Message `974`.

11.7 Migrating to 64 bits

Applications written for the traditional 32-bit model where `int`, `long` and pointers are each represented in 32 bits, may have difficulty when ported to one of the 64-bit models. Problems that you may encounter and that PC-lint Plus will catch are described and implemented as options in file `au-64.lnt`. This file and other `.lnt` files mentioned below are distributed with the product and/or are downloadable from our web site.

The file `au-64.lnt` is not intended to be used directly by the programmer. There are a number of wrappers reflecting the different flavors of 64-bit computing. These wrappers specify sizes and other options unique to specific models and then invoke `au-64.lnt`. The models and the `au` file that you should be using are described below.

Data Type	LP64 Model	LLP64 Model	ILP64 Model
long long	64 bits	64 bits	64 bits
pointers	64 bits	64 bits	64 bits
long	64 bits	32 bits	64 bits
int	32 bits	32 bits	64 bits

The table above shows the differences between each of the 3 64-bit models. Each model has a corresponding au file: `au-lp64.lnt` for LP64, `au-llp64.lnt` for LLP64, and `au-ilp64.lnt` for ILP64.

11.8 Deprecation of Entities

You may indicate that a particular *name* is not to be employed in your programs by using this option:

`-deprecate(`*`category`*` , `*`name`*` [,`*`commentary`*`])`

category is one of: `function`, `keyword`, `macro`, `option`, `variable`, `type`, `basetype`, `printf_code` or `scanf_code`.

The *commentary* in the third argument will be appended to the message. For example,

```
-deprecate( variable, errno, Violates Policy XX-123 )
```

When the use of `errno` as a variable is detected (but not its definition or declaration) the following Warning is issued.

```
Warning 586: variable 'errno' is deprecated. Violates Policy XX-123
```

When the category of deprecation is `variable` only the use of external variables are flagged. Local variables may be employed without disparaging comment.

If `errno` were a `macro` you would need to deprecate `errno` as a `macro`:

```
-deprecate( macro, errno, Violates Policy XX-123 )
```

If `errno` could be either (the standard allows both forms) then both options should be used.

You may also deprecate functions and keywords. For example:

```
-deprecate( keyword, goto, goto is considered harmful )
-deprecate( function, strcpy, has been known to cause overuns )
```

could be used to flag the use of suspect features.

Quotes (both single and double) and parentheses within the commentary need to be balanced.

11.8.1 Deprecation of Options

Options can also be deprecated. Deprecating an option causes future uses of that option to be met with message 586 although the option is still processed as usual. When deprecating an option, you must specify the name of the option, including a leading '-' or '+' but must not provide any arguments for the option. For example:

```
-deprecate( option, -setenv, environment variables should not be set during the
linting process)
```

will deprecate the use of `-setenv`. It is not possible to deprecate the use of **individual** flag options; using the `-deprecate` option with `-f`, `+f`, `--f`, or `++f` will deprecate **all** flag options. Note that some options have forms that begin with '-' and another form that begins with '+'; deprecating one form does not automatically cause the other form to be deprecated even if both forms have identical meanings.

11.8.2 Deprecation of Types

The `-deprecate` option can be used to deprecate types. This can be accomplished using the deprecation categories '`type`' and '`basetype`'.

When the category '`type`' is specified, message 586 is issued for any use of the type in a declaration but type alias names (introduced via `typedef` or `using`) are not looked through and use of the underlying type is allowed without complaint if it occurs through such an alias.

The category of '`basetype`' is similar except type aliases are looked through and if at any level the deprecated type is present, 586 is issued. If the deprecated type is a `typedef` type, no diagnostic is issued for the declaration of the type (although use of the type is diagnosed).

Using a deprecated `type` as a target of a `typedef` is not diagnosed with 586. The logic is that for '`basetype`', the use of the `typedef` that targets the deprecated type will be diagnosed anyway and that for '`type`' the user is not interested in use of the type through `typedef`s. We do provide a new elective message, 986, that will be issued when the target of a type alias is deprecated with the '`type`' category.

```
//lint -deprecate(type, int) warn about uses of int but not through an alias
//lint -deprecate(basetype, DOUBLE)

typedef int INT;         // 986 issued for 'int' appearing in typedef
typedef double DOUBLE;   // okay, declaration of deprecated type
typedef DOUBLE DOUBLE2;  // okay, use of deprecated basetype in alias

int i;// 586 - 'int' is deprecated
INT i2;// okay, use of deprecated type via alias

union u {
    const int * pci;    // 586 - deprecated type 'int' used in declaration
```

```
    double * pd;        // okay, 'double' is not deprecated
    DOUBLE d1;          // 586 - deprecated basetype 'DOUBLE' used directly
    DOUBLE2 d2;         // 586 - deprecated basetype 'DOUBLE' used indirectly
};
```

11.8.3 Deprecation of Format Function Conversion Specifiers

The `-deprecate` option can be used to deprecate conversion specifiers for `printf`-like and `scanf`-like functions using the `printf_code` and `scanf_code` categories. For example,

```
-deprecate(printf_code, n)
```

will deprecate the use of `%n` in `printf`-like functions, note that the `%` is not included in the `-deprecate` option. Deprecating a conversion specifier will result in message 586 being issued if the conversion specifier is used, regardless of any length modifiers present in the actual use, but will not deprecate other conversion specifiers with the same meaning. For example `-deprecate(printf_code, i)` will warn for `%i` and `%hi` but will not warn for `%d` (which has the same meaning as `%i` in `printf`-like functions).

11.9 Parallel Analysis

PC-lint Plus supports the long-requested feature of utilizing multiple cores to achieve faster processing times. This feature is enabled by placing the new `-max_threads=`*n* option before the first module to process. If this option does not appear before the first module is seen, the behavior is as if `-max_threads=1` was used. Threads are used both during the main processing phase and the global wrap-up phase. In the main phase, a separate thread is dispatched to handle each module, up to a maximum of *n* concurrent threads. When all of the modules have been processed, threads are employed to handle wrap-up processing, again up to a maximum of *n* concurrent threads.

While results will vary depending on a variety of factors, the best overall times are typically achieved when using a value for *n* that equals the number of available cores, or about twice the number of cores for processors that support hyper-threading. Some experimentation may be necessary to find the best value for *n* on a particular system. In order to assist in that regard, the option `-max_threads=0` will result in PC-lint Plus picking a value for *n* that it thinks is optimal based on querying of the available hardware for systems that support it. When using `-max_threads=0`, elective note 999 will report on the number of threads that has been selected.

There are a few caveats to keep in mind when employing multiple threads:

1. Very little memory is shared between threads, which means that memory usage scales roughly linearly with the number of concurrent threads. For example, if using 1 thread results in memory usage of 500MB, it probably wouldn't be productive to utilize more than 4 threads on a system with 2GB of RAM, regardless of how many cores may be available.

2. Output is buffered by module when using multiple threads. This means that the output for a module will not be emitted until the entire module is processed (this happens before global wrap-up). Additionally, the order in which modules are processed is not guaranteed although output will never be interleaved between modules. For example, when processing modules `A` and `B`, the diagnostics for module `A` may appear (in their entirety) before or after module `B` when using multiple threads and this order may change between runs. When using a single thread, diagnostics for one module will always appear before the diagnostics for a later-provided module.

3. Interactive features such as the Value Tracking Debugger are supported only when executing with a single thread.

Aside from the above caveats, there is no difference in behavior or functionality when using multiple threads.

11.10 Language Limits

The C and C++ Standards define minimum translation limits that must be supported by a conforming compiler. The limits specify quantities such as the minimum number of significant characters in internal and external identifiers, the minimum number of function parameters that an implementation must support, the minimum number of supported concurrently defined macros, and the minimum number of data members supported in structures. The C99 limits are specified in section 5.4.2.1 of the C99 Standard (ISO/IEC 9899:1999) and the C++ limits are specified in Annex B of the C++ Standard (ISO/IEC 14882:2011).

`-lang_limit(C\C++, *limit-name*, *limit-value*)` specify minimum language translation limits

> The `-lang_limit` option takes three arguments. The first argument is either C or C++ indicating which language the overridden limit applies to. The second argument is the type of limit and must be one of the names in the first column of the above table. The last argument is a value that must be between 0 and 4294967294 (0 indicates the lack of a limit) or the special value of `default`, which reverts back to the corresponding value of the below table essentially removing a previously overridden value.

The table below lists the language limit checks supported by PC-lint Plus. The Limit Name is the name recognized in the second parameter of the `-lang_limit` option. The Limit Description is the text that is used in the `793` message when the limit is exceeded and which can be used with the `-estring` option for suppression purposes. The C89 Limit shows the minimum limits specified by the ANSI C89 standard. The C99 Limit column shows the minimum limits mandated by C99, the C11 Limits are identical. The C++ Limit shows the limits required by the C++ Standard (all versions of C++ share the same limits).

Limit Name	Limit Description	C89 Limit	C99 Limit	C++ Limit
`external_identifiers`	external identifiers	511	4095	65526
`internal_identifier_chars`	significant characters in an internal identifier	31	63	1024
`macro_identifier_chars`	significant characters in a macro name	31	63	1024
`external_identifier_chars`	significant characters in an external identifier	6	31	1024
`function_parameters`	function parameters	31	127	256
`function_arguments`	function arguments	31	127	256
`macro_parameters`	macro parameters	31	127	256
`string_literal_length`	characters in a string literal	509	4095	65536
`case_labels`	case labels in a switch	257	1023	16384
`structure_members`	structure members	127	1023	16384
`enumeration_constants`	enumeration constants	127	1023	4096
`base_classes`	base classes	n/a	n/a	16384
`direct_base_classes`	direct base classes	n/a	n/a	1024
`class_members`	members in a class	n/a	n/a	4096
`static_members`	static members in a class	n/a	n/a	1024
`final_functions`	final overriding virtual functions in a class	n/a	n/a	16384
`virtual_base_classes`	virtual base classes	n/a	n/a	1024
`friend_decls`	friend declarations in a class	n/a	n/a	4096
`access_decls`	access control declarations in a class	n/a	n/a	4096
`ctor_initializers`	member initializers in a constructor	n/a	n/a	6144
`scope_qualifiers`	scope qualifiers in an identifier	n/a	n/a	256
`template_arguments`	template arguments in a template	n/a	n/a	1024
`try_handlers`	handlers in a try block	n/a	n/a	256
`throw_specs`	throw specifications in a function	n/a	n/a	256

By default, the limits shown above are used to determine when a minimum limit has been exceeded and, for C, is dependent on the version of the language used. If your compiler supports different limits, or if you just want to be alerted when a different threshold is reached for a particular limit, you can use the `-lang_limit` to override the defaults shown above.

The below table provides additional details about some of the limits checked by PC-lint Plus.

Limit Name	Notes
`external_identifiers`	External identifiers are functions and variables with external linkage.
`internal_identifiers`	Internal identifiers are any non-preprocessor (e.g. macro) symbols that are not external identifiers. These include type names, class names, local variables, enumeration constants, etc.
`case_labels`	This does not include the `default` label or case labels of nested switches.
`structure_members`	This includes only non-static data members but includes members inherited from base classes.
`base_classes`	The number of bases for a given class including indirect and virtual bases.
`direct_base_classes`	Virtual and non-virtual direct base classes for a class.
`class_members`	Includes all static and non-static data and function members declared directly in the class (e.g. not inherited members). Note that this also includes implicitly generated functions such as constructors, assignment operators, etc.
`static_members`	All static data and function members, including inherited members for a class.
`virtual_base_classes`	Direct and indirect virtual base classes for a class.
`access_decls`	The number of access control (or using) declarations present in a class. Does not include base classes. This is not a count of the *access-specifiers* present in a class.
`ctor_initializers`	The number of items initialized in a constructor member initializer list. Includes base class initializers.
`scope_qualifiers`	This is the number of nested name specifiers present in a *qualified-id*, e.g. `A::B::C` contains two scope qualifiers.

12 Preprocessor

12.1 Preprocessor Symbols

PC-lint Plus supports several predefined macros including those defined by ISO C and those supported by various compilers. The special behavior of any of these macros will be removed for the remainder of the module if the macros are explicitly defined or undefined using `#define` or `#undef` and permanently removed if defined of undefined with the `-d`/`+d`/`++d` or `-u`/`--u`.

- `_lint` – The special preprocessor symbol `_lint` is pre-defined with a value representing the version of PC-lint Plus. The primary purpose of this symbol is to enable the programmer to determine whether PC-lint Plus is processing the file.

  For example, if you have a section of code that is unacceptable to PC-lint Plus for some reason (such as in-line assembly code), you can use `_lint` to make sure that PC-lint Plus doesn't see it. Thus,

  ```
  #ifndef _lint
  ...
  Unacceptable coding sequence
  ...
  #endif
  ```

  will cause PC-lint Plus to skip over the elided material.

  The value of `_lint` is 1000 * Major Version Number + 10 * Minor Version number + the Patch Level.

Version	*Value* of `_lint`
1.0.0	1000
1.0.1	1001
1.1.0	1010
1.2.1	1021
1.12.3	1123
2.0.0	2000

  E.g.

  ```
  #if _lint >= 900
      // use Version 9 feature
  #endif
  #if _lint != 902
      // not for Version 9.02
  #endif
  ```

- `__cplusplus` – This symbol is defined for each module that is interpreted as being a C++ module and is otherwise undefined. The value that this symbol expands to is dependent on the C++ language mode: the value is 201402L for C++14, 201103L for C++11, and 199711L for C++03. C++ modules are determined by extension and possibly by option. See Chapter 3 The Command Line.

- `__COUNTER__` – This macro expands to an integer that automatically increments every time the macro is expanded within a module. The first result of the first expansion in a module is 0, the second expansion is 1, etc. When used with the `##` operator, this macro provides a mechanism to generate unique identifiers.

- `__BASE_FILE__` – Expands to a string literal that contains the name of the module being processed, as the name was provided to PC-lint Plus.

- `__INCLUDE_LEVEL__` – Expands to a non-negative integer representing the `#include` nesting depth in which the macro appears. A value of 0 indicates a non-header location.
- `__TIMESTAMP__` – Expands to a string literal that contains the last modification date and time of the file in which the macro appears, as returned by the `asctime` function. If the modification time information cannot be determined, expands to the string literal `"??? ??? ?? ??:??:?? ????"`.
- The following pre-defined identifiers begin and end with double underscore and are ANSI/ISO compatible.

  `__TIME__` – The current time
  `__DATE__` – The current date
  `__FILE__` – The current file
  `__LINE__` – The current line number
  `__STDC__` – Defined to be 1.
  `__STDC_VERSION__` – This is undefined for C++. It is defined for C by default as '`199901L`'. If you select an earlier version of C using `-A` option as in `-A(C90)` this will be undefined.
  `__STDC_HOSTED__` – This is defined whenever `__STDC_VERSION__` is defined. When defined it is defined to be 0.

  Compiler-dependent preprocessor symbols may also be established as described in Section 4.11 Compiler Adaptation.

12.2 #include Processing

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

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

   `-i`*`directory`*

   in the order in which the options were presented. If this fails we go to step 3.
3. On systems supporting environment variables, each directory in the sequence of directives specified by the `INCLUDE` environment variable is prepended to the file.
4. There is an attempt to `fopen` the file by the name provided, without considering flag `fdi`.

If the include directive is of the form

`#include <`*`filename`*`>`

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

12.2.1 INCLUDE Environment Variable

The `INCLUDE` environment variable may specify a search path in which to search for header files (`#include` files). For example:

```
set INCLUDE=b:\include;d:\extra
```

specifies that, should the search for a `#include` file within the current directory fail, a search will be made in the directory `b:\include` and, on failing that, a search will be made in the directory `d:\extra`. This searching is done for modules as well as `#include` files. You may select an environment variable other than `INCLUDE`. See the `-incvar` option.
Notes:

1. No blank may appear between 'INCLUDE' and '='. Blanks adjacent to semicolons (;) are ignored. All other blanks are significant

2. A terminating semi-colon is ignored.

3. This facility is in addition to the -i... option and is provided for compatibility with a number of compilers in the MS-DOS environment.

4. Any directory specified by a -i directive takes precedence over the directories specified via the INCLUDE environment variable.

12.3 ANSI/ISO Preprocessor Facilities

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

12.3.1 #line and

A C/C++ preprocessor may place #line directives within C/C++ source code so that compilers (and other static analyzers such as PC-lint Plus) can know the original file and original line numbers that produced the text actually being read. In this way, these processors can report errors in terms of the original file rather than in terms of the intermediate text.

By default, #line directives are processed. To ignore #line directives use the option -fln. Some systems support # as an abbreviation for #line; these are treated equivalently by PC-lint Plus.

12.4 Non-Standard Preprocessing

Preprocessor commands in this section need to be activated via the +ppw option. Also, their semantics may be copied via the ppw_asgn option.

12.4.1 #import

This preprocessor directive is intended to support the Microsoft preprocessor directive of the same name. For example:

```
#import "c:\compiler\bin\x.lib"}
```

will determine the base name (in this case "x") and attempt to include, as a header file, *basename*.tlh. Thus, for linting purposes, this directive is equivalent to:

```
#include "x.tlh"
```

Options that may accompany #import are ignored. When the (Microsoft) compiler encounters a #import directive it will generate an appropriate .tlh file if a current one does not already exist. PC-lint Plus will not generate this file.

When compiling, it is possible to place the generated .tlh file in a directory other than the directory of the importing file. If this option is chosen, then when linting, this other directory needs to be identified with a -i option or equivalent.

This preprocessor word is not enabled by default. It can be enabled via the +ppw(import) option. This option has been placed into the various compiler options files for the Microsoft C/C++ compiler.

12.4.2 #include_next

`#include_next` is supported for compatibility with the GNU C/C++ compiler. It uses the same arguments as `#include` but starts the header file search in the directory just after the directory (in search order sequence) in which the including file was found. See Section 12.2 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
```

12.4.3 #ident

This directive takes a single argument, a string constant. This directive will cause some compilers to copy the string into an implementation defined portion of the resulting object file. PC-lint Plus processes but ignores this directive.

12.4.4 #sccs

This is treated identically to `#ident`.

12.4.5 #warning

The `#warning` directive is used by some compilers to emit a user-defined warning when the directive is reached during preprocessing. It is similar to `#error` but doesn't terminate processing. If this keyword is enabled, PC-lint Plus will issue warning 490 along with the contents of the line that follows the `#warning` directive, in the same manner as for `#error`. In particular, the text that follows is emitted as written except that multiple space characters are collapsed into a single space and macros are not expanded. The text does not need to be a string constant. If your compiler calls this directive `#warn`, you can use

```
+ppw(warning)
-ppw_asgn(warn, warning)
```

to cause PC-lint Plus to support the alternate spelling.

12.5 User-Defined Keywords

PC-lint Plus might stumble over strange preprocessor commands that your compiler happens to support. For example, some Unix system compilers support `#assert`. Since this is something that can NOT be handled by a suitable `#define` of some identifier we have added the `+ppw(`*command-name*`)` option ('`ppw`' is an abbreviation for PreProcessor Word). For example, `+ppw(ident)` will add the preprocessor command alluded to above. PC-lint Plus recognizes and ignores the construct.

12.6 Preprocessor sizeof

The non-standard use of `sizeof` in a preprocessor conditional is supported by some older and embedded compilers but not directly supported by PC-lint Plus because the information necessary to evaluate a `sizeof` is not available during the preprocessing phase. For portability reasons, such constructs should not be used in new code, favoring static assertions or similar mechanisms instead.

For legacy code, we provide a work-around using the new `-pp_sizeof(`*Text*`, `*Value*`)` option that can be used to direct PC-lint Plus on how to evaluate a particular `sizeof` expression appearing in a preprocessor conditional. *Text* is the text of the expression appearing inside of the `sizeof` and `Value` is the integral value to apply to the evaluation of the `sizeof` expression. The new warning 2491 is issued when a preprocessor `sizeof` is encountered with an expression that has not been registered with `-pp_sizeof`. This message can be used to identify expressions that need to be registered as well as the text to use for *Text*. For example:

```
#if sizeof(int) < 4
#error "int is too small"
#endif
```

will elicit:

```
test_ppsizeof.c  1  warning 677: sizeof used within preprocessor statement
#if sizeof(int) < 4
    ^
test_ppsizeof.c  1  warning 2491: unknown expression 'int' in sizeof will
    evaluate to 0, use -pp_sizeof to change the value used for evaluation
test_ppsizeof.c  2  error 309: #error "int is too small"
#error "int is too small"
 ^
```

The 677 message warns of the use of `sizeof` in the preprocessor, warning 2491 alerts the programmer that PC-lint Plus doesn't know how to handle `sizeof(int)` in the preprocessor and provides direction for using the `-pp_sizeof` option. Message 309 is issued for the failed `#error` directive resulting from the fact that the unknown `sizeof` expression was evaluated to be 0.

To employ the work-around, determine what the value of `sizeof(int)` is on the target platform and register the expression with `-pp_sizeof`. E.g. if `int` is 4 bytes, use `-pp_sizeof(int, 4)`, which will cause the `sizeof` expression to be evaluated as expected. A separate `-pp_sizeof` option must be used for each expression that will appear inside of a preprocessor `sizeof`.

13 Living with Lint

(or Don't Kill the Messenger)

The comments in this chapter are suggestive and subjective. They are the thoughts and opinions of only one person and for this reason are written in the first person.

When you first apply PC-lint Plus against a large C or C++ program that has not previously been linted, you will no doubt receive many more messages than you bargained for. You will perhaps feel as I felt when I first ran a Lint against a program of my own and saw how it rejected 'perfectly good' C code; I felt I wanted to write in C, not in Lint.

Stories of Lint's effectiveness, however, are legendary. PC-lint was, of course, passed through itself and a number of subtle errors were revealed (and continue to be revealed) in spite of exhaustive prior testing. I tested a public domain grep that I never dared use because it would mysteriously bomb. PC-lint found the problem - an uninitialized pointer.

It is not only necessary to test a program once but it should be continuously tested throughout a development/maintenance effort. Early in Lint's development we spent a considerable effort, over several days, trying to track down a bug that Lint would have detected easily. We learned our lesson and were never again tempted to debug code before linting.

But what do you do about the mountain of messages? Separating wheat from chaff can be odious especially if done on a continuing basis. The best thing to do is to adopt a policy (a policy that initially might be quite liberal) of what messages you're happy to live without. For example, you can inhibit all informational messages with the option `-w2` . Then work to correct only the issues associated with the messages that remain. DO NOT simply suppress all warnings with something like: `-e*` or `-w0` as this can disguise hard errors and make subsequent diagnosis very difficult. The policy can be automatically imposed by incorporating the error suppression options in a `.lnt` file (examples shown below) and it can gradually be strengthened as time or experience dictate.

Experience has shown that linting at full strength is best applied to new programs or new subroutines for old programs. The reasons for this is that the various decisions that a programmer has made are still fresh in mind and there is less hesitancy to change since there has been much less 'debugging investment' in the current design. Decisions such as, for example, which objects should be `signed` and which `unsigned`, can benefit from checking at full strength. Full strength can even mean torture testing (see Torture Testing Your Code).

13.1 An Example of a Policy

An example of a set of practices with which I myself can comfortably live, is as follows.

Mistaking assignment for equality is a potential problem for C/C++. If a Boolean test is made of assignment as in

```
if( a = f( ) ) ...
```

PC-lint Plus will complain with message 720. If the assignment is wrapped with parentheses as in

```
if( (a = f()) ) ...
```

a different message (820) is used. This is done deliberately so that the programmer may distinguish between a conscious request and what appears to be accidental. Combining the assignment with testing is such a useful operation that I'm happy to put up with an extra pair of parentheses. Therefore, I suppress 820 with the option

```
-e820
```

At one time I mixed `unsigned` and `signed` quantities with almost reckless abandon. I now have considerably more respect for the subtle nuances of these two flavors of integer and now follow a more cautious approach. I had previously employed the options

```
-e713 -e737
```

(713 involves assigning `unsigned` to `signed`, 737 is loss of sign). These inhibitions affect only some variation of assignment. We retain warnings about mixing `signed`/`unsigned` with binary operators.

I also no longer think it is a great idea to automatically inhibit 734 (sub-integer loss of precision). This message can catch all sorts of things such as assigning `int` to `short` when `int` is larger than `short`, assigning oversize `int` to `char`, assigning too large quantities into bit fields, etc.

I suppress messages about shifting `int` (and `long`) to the left but I want to be notified when they are shifted right as this can be machine-dependent and is generally regarded as a useless and hazardous activity. Therefore, I use `-e701 -e703`.

I want to run my code through at least two passes so that cross-functional checks can be made. The option is `-vt_passes(2)`.

I place my list of favorite error-suppression options in a file called `options.lnt`. It looks like this:

```
options.lnt:
-e820            // parenthesized test of assignment
-e701 -e703      // shifting int left is OK
-vt_passes(2)    // use two value tracking passes
```

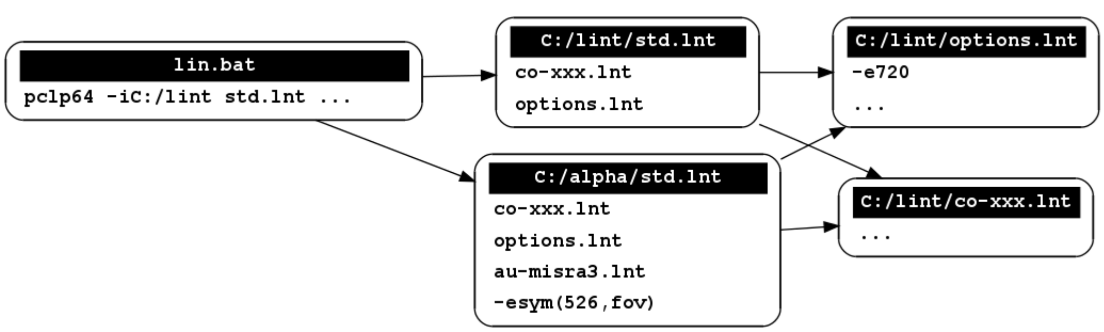
13.2 Recommended Setup

The recommended setup includes a default standard configuration (`std.lnt`) and a shell script (`lin.bat` or `lin.sh`) that invokes PC-lint Plus with `std.lnt`. The shell script might look like:

```
pclp64 -iC:/lint std.lnt ...
```

The "`lin`" script needs to be placed in your executable path.

When run from most directories, the file `std.lnt` is found in the PC-lint Plus directory (owing to the `-i` option appearing in the script). This in turn includes a compiler options file and a centralized `options.lnt` as shown below, also found in the PC-lint Plus directory. When run from a directory that has its own `std.lnt` file however, the local `std.lnt` overrides the standard one. In this way, each project can maintain its own configuration and running PC-lint Plus from a directory that doesn't have an explicit configuration will result in the default standard configuration being applied.



13.3 Final Thoughts

In summary, establish procedures whereby PC-lint Plus may be conveniently accessed for a variety of purposes. Lint small pieces of a project before doing the whole thing. Establish an error-message suppression policy that may initially be somewhat relaxed and can be strengthened in time. Lint full strength on new projects. But don't kill the messenger!

14 Common Problems

14.1 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!
```

14.2 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.

14.3 Too many messages

It should be emphasized that suppressing a message does not alter the behavior of PC-lint Plus other than to suppress the message. For example, inhibiting message **718** (function used without a prior declaration) does not inhibit other messages about the function such as "inconsistent return value" or "inconsistent parameters". It is as if you had edited the output file and removed all references to message **718**. So you needn't be too hesitant about inhibiting a message.

To set a warning level, use option `-w`, or `-wlib` (See Section 4.3.1 Error Inhibition.)

14.4 What is the preprocessor doing?

In order to understand some diagnostics it may be necessary to look at the output of the preprocessor. The option `-p` turns PC-lint Plus into a preprocessor. For example:

```
in -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

```
lint -v1 -p alpha.cpp
```

This will provide a line-by-line description as to how the headers relate to the output.

14.5 Plain Vanilla Functions

N.B. The following pertains only to C code.

By a plain vanilla function (or canonical function) we mean a function declared without a prototype. For example

```
void f();
```

Not too many programmers realize that such a function is incompatible with one that is prototyped with a `char`, `short`, or `float` parameter or has an ellipsis. We warn you (type difference = '`promotion`' or '`ellipsis`') but the warning can cause confusion if you do not realize the difference.

When a call is made to such a function the compiler must decide which, if any, promotions to apply to the arguments. Since the declaration said nothing about arguments, a standard (i.e., canonical) set of promotions is applied. According to ANSI, `char` and `short` are promoted to `int`, and `float` is promoted to double. Also the argument list is presumed fixed so that registers may be used to pass arguments.

Prototypes can inhibit such promotions; if f was declared:

```
void f( char, short, float );
```

All three promotions would be inhibited. For this reason this declaration is incompatible with the earlier declaration and you receive a warning. If `f` were declared:

```
void f( int, ...);
```

we again warn you because the canonical declaration allows the compiler to pass arguments in registers and the ellipsis forces the compiler to pass arguments on the stack.

This is all in the ANSI/ISO C standard.

14.6 Avoiding Lint Comments in Your Code

Occasionally there is a requirement that there be no lint directives in your source code. A programmer can go pretty far in inhibiting unwanted messages by using the `-e`, `-esym`, etc. options placed within a `.lnt` file but occasionally it happens that a specific occurrence of a message needs to be suppressed. For example:

```
int *pi;
unsigned **ppu;
pi = (int *) ppu;
```

raises message **740** (unusual pointer cast). To suppress this particular usage you may define a macro as in:

```
#define INT_STAR(p) ((int *) (p))
 ...
pi = INT_STAR(ppu);
```

You will still get the message, which can then be suppressed with

```
-emacro(740,INT_STAR)
```

14.7 Strange Compilers

You may want to lint programs that have been prepared for compilers that accept strange and unusual constructs, and for which we do not provide a custom compiler options file. There are a number of options you can use to get PC-lint Plus to ignore such constructs. Chief among these are the `-d`, `+rw` and `+ppw` options. But also check Section 4.11.1 Customization Facilities for additional options to help cope with the truly extraordinary.

14.8 !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)
```

14.9 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=
```

14.10 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 Plus will stumble over this construct yielding inappropriate messages. To get PC-lint Plus to ignore such statements, you may define `EXECSQL` to be equivalent to the built-in reserved word `_to_semi` (See Section 4.11.1 Customization Facilities). Do not forget to activate the reserved word. The pair of options needed are:

```
-dEXECSQL=_to_semi
+rw(_to_semi)
```

14.11 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 Plus. There are several things you can do:

`+fsc`	(String constants are const char flag) assumes string constants are `const char*`.
`-vt_passes=10 -vt_depth=10`	Make sure you use plenty of passes as a sequence of calls can have a ripple effect.
`+fpn`	(Pointer parameter may be Null flag) warns about the use of pointer parameters without first checking for NULL.
`+fnr`	(Null can be Returned flag) Any pointer returned by any function is assumed to possibly be Null
`-strong(AJX)`	All typedefs must match exactly.
`-w4`	Set warning to the max. (This will probably be more torture than you can take - you've been warned).

14.12 Cautions with make

Users of '`make`' programs may be surprised to see Lint output something like:

```
--- Module: /usr/local/bin/pclp64_linux
```

This happens when the `make` target for Linting contains:

```
\lint_proj:
    $(LINT) $(LINT_OPTIONS) $(INCLUDES) $(SOURCES) > $(LINT_LOG)
```

Users of 'make' might assume the make variable 'LINT' should hold the path to the Lint executable; it follows in the tradition of other standard make variable names like 'CC', 'CXX', etc. The problem is that make variables may also be environment variables, and Lint uses the environment variable 'LINT' as a prefix to the list of command line arguments (See Chapter 4.). The solution is simply to use some variable name other than 'LINT' (e.g. 'LINT_EXE').

15 Messages

Every diagnostic message has an associated message number. By looking up the number in the list below you can obtain additional information about the cause of the diagnostic. This information is also available as a machine-readable ASCII file `msg.txt` as well as JSON (`msg.json`).

Messages pertaining specifically to C++ generally reside in the 1xxx and 3xxx ranges while those applicable to C or both C and C++ reside in the xxx and 2xxx ranges. After a possible 1000 is subtracted off, the remainder lies in the range 0-999.

Remainders in the range 1-399 are errors that may be the result of incorrect syntax, a violation of the semantics of the language or a processing error. Some serious errors can be fatal, unsuppressible, or both. Errors are generally mistakes in the program or the PC-lint Plus configuration and should be corrected. While most errors can be suppressed, it is not advisable to do so unless the root cause is understood and it is determined that suppression is both safe and the only available course of action. Suppressing error messages can hide configuration issues that may result in incorrect or incomplete analysis.

Remainders in the 400-699 range are warning messages that indicate that something is potentially wrong with the program being examined. Remainders in the range 700-899 designate informational messages; these messages serve to point out unusual constructs, generally acknowledged bad practices, and potential pitfalls.

Remainders in the range 900-999 are called "Elective Notes" and diagnose specific points of interest that do not necessarily represent any deficiency in the code but may be of interest to certain users. They are not automatically presented. You may examine the list to see if you wish to be alerted to any of them.

Aside from the 4 ranges mentioned above, the 4xxx and 5xxx ranges are used for C and C++ error messages, the 8xxx range is reserved for user-defined messages, and the 9xxx ranges is used for C and C++ elective notes. Messages in the 6xxx and 7xxx ranges are reserved for future use.

Note that there are roughly 2000 clang compiler errors that PC-lint Plus reports in the 4xxx and 5xxx range. Because these messages are generally self-explanatory error messages, they are not included in this document.

Range	Description	Warning Level
1-199	C Syntax Errors	1
200-299	Internal Errors	1
300-399	Fatal Errors	1
400-699	C Warnings	2
700-899	C Informational	3
900-999	C Elective Notes	4
1000-1199	C++ Syntax Errors	1
1200-1299	Internal Errors	1
1300-1399	C++ Fatal Errors	1
1400-1699	C++ Warnings	2
1700-1899	C++ Informational	3
1900-1999	C++ Elective Notes	4
2000-2199	C Syntax Errors	1
2200-2399	Reserved	
2400-2699	C Warnings	2
2700-2899	C Informational	3
2900-2999	C Elective Notes	4
3000-3199	C++ Syntax Errors	1
3200-3399	Reserved	
3400-3699	C++ Warnings	2
3700-3899	C++ Informational	3
3900-3999	C++ Elective Notes	4
4000-5999	C and C++ Errors	1
6000-6999	Reserved	
7000-7999	Reserved	
8000-8999	User Defined	3
9000-9999	Misc Elective Notes	4

Glossary

A few of the terms used in the commentary below are:

argument The actual argument of a function as opposed to a dummy (or formal) parameter of a function (see *parameter* below).

arithmetic Any of the integral types (see below) plus `float`, `double`, and `long double`.

Boolean In general, the word Boolean refers to quantities that can be either true or false. An expression is said to be Boolean (perhaps it would be better to say 'definitely Boolean') if it is of the form: `operand op operand` where `op` is a relational (`> >= < <=`), an equality operator (`== !=`), logical *And* (`&&`) or logical Or (`||`). A context is said to require a Boolean if it is used in an `if` or `while` clause or if it is the 2nd expression of a `for` clause or if it is an argument to one of the operators: `&&` or `||`. An expression needn't be definitely Boolean to be acceptable in a context that requires a Boolean. Any integer or pointer is acceptable.

declaration Gives properties about an object or function (as opposed to a definition).

definition that which allocates space for an object or function (as opposed to a declaration) and that may also indicate properties about the object. There should be only one definition for an object but there may be many declarations.

integral a type that has properties similar to integers. These include `char`, `short`, `int`, and `long` and the `unsigned` variations of any of these.

scalar any of the arithmetic types plus pointers.

lvalue is an expression that can be used on the Left hand side of an assignment operator (=). Some contexts require lvalues such as autoincrement (++) and autodecrement (--).

macro an abbreviation defined by a `#define` statement. It may or may not have arguments.

member elements of a `struct` and of a `union` are called members.

module that which is compiled by a compiler in a single independent compilation. It typically includes all the text of a `.c` (or a `.cpp` or `.cxx`, etc.) file plus any text within any `#include` file(s).

parameter A formal parameter of a function as opposed to an actual argument (see `argument` above).

15.1 Message Parameters

Most messages are parameterized with one or more pieces of information that may be different each time the message is issued such as the name of a symbol being referenced, the types involved in a conversion, or a string representing dynamic text. These parameters can be employed for suppression purposes using `-esym`, `-estring`, and `-etype`.

There are three categories of message parameters: *Strings*, *Symbols*, and *Types*. *Symbol* parameters in a message are represented as *symbol* in the message descriptions below and may be used with the `-esym` option to suppress the message. *Type* parameters are represented as *type* and may be used with `-etype` to suppress the message. `-etype` may also be used to suppress *symbol* parameters by using the type of a symbol appearing in the message (see `+typename` for information about obtaining a type string suitable for use with `-etype` in this fashion). *String* parameters consist of virtually every other italicized parameter including:

- context
- detail
- file
- integer
- name
- operator
- string
- strong-type

Additionally, some messages contain dynamic text, which is represented by a slash in the message such as:

```
444: for statement condition tests incremented/decremented pointer for null
```

`-estring` can be used to suppress such messages when one of these values is used in the message, e.g. `-estring(444, incremented)`. `-esym` can also be used to suppress on *String* parameters when the `fsn` flag is ON (which it is by default).

Detailed parameter information for specific instances of messages can be obtained by using `+paraminfo`.

15.2 Messages 1-999

1 unclosed comment
error End of file was reached with an open comment still unclosed.

2 unclosed quote
error An end of line was reached and a matching quote character (single or double) to an earlier quote character on the same line was not found.

3 #elif without a #if
error A `#elif` was encountered not in the scope of a `#if`, `#ifdef` or `#ifndef`.

5 **too many #endif directives**
error
A `#endif` was encountered not in the scope of a `#if` or `#ifdef` or `#ifndef`.

8 **unclosed #if**
error
A `#if` (or `#ifdef` or `#ifndef`) was encountered without a corresponding `#endif`.
Supports MISRA C 2004 Rule 19.17 (Req)
Supports MISRA C 2012 Rule 20.14 (Req)
Supports MISRA C++ Rule 16-1-2 (Req)

9 **#elif after #else**
error
A given `#if` contained a `#else`, which in turn was followed by either another `#else` or a `#elif`. The error message gives the line of the `#if` statement that started the conditional that contained the aberration.

10 **expecting *detail***
error
string is the expected token. The expected token could not be found. This is commonly given when certain reserved words are not recognized. For example:

```
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 Section 14.7 Strange Compilers.

12 **need < or "**
error
After a `#include` is detected and after macro substitution is performed, a file specification of the form <*filename*> or "*filename*" is expected.
Supports MISRA C 2004 Rule 19.3 (Req)
Supports MISRA C 2012 Rule 20.3 (Req)
Supports MISRA C++ Rule 16-2-6 (Req)

13 **'*string*' cannot be signed or unsigned**
error
A type adjective such as `long`, `unsigned`, etc. cannot be applied to the type, which follows.

15 **symbol *symbol* redeclared (*type* vs. *type*)**
error
The named symbol has been previously declared or defined in some other module (location given) with a type different from the type given by the declaration at the current location. The parenthesized type parameters provide the two differing types.
Supports MISRA C 2004 Rule 8.4 (Req)
Supports MISRA C 2012 Rule 8.4 (Req)
Supports MISRA C++ Rule 3-2-2 (Req)
Supports MISRA C++ Rule 3-2-4 (Req)

16 **unknown preprocessor directive**
error
A `#` directive is not followed by a recognizable word. If this is not an error, use the `+ppw` option. (Section 4.11 Compiler Adaptation).
Supports MISRA C 2004 Rule 19.16 (Req)

Supports MISRA C 2012 Rule 20.13 (Req)
Supports MISRA C++ Rule 16-0-8 (Req)

18 symbol *symbol* redeclared (*typediff*)
error
A symbol is being redeclared. The parameter *typediff* provides further information on how the types differ.
Supports MISRA C 2004 Rule 8.4 (Req)
Supports MISRA C++ Rule 2-10-6 (Req)
Supports MISRA C++ Rule 3-2-1 (Req)

19 C++ requires a type specifier for all declarations
error
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;;
```

21 array initializer must be an initializer list
error
An initializer for an indefinite size array must begin with a left brace.

24 expected an expression
error
An operator was found at the start of an expression but it was not a unary operator.

25 character constant too long for its type
error
Too many characters were encountered in a character constant (a constant bounded by `’` marks).

29 duplicated type-specifier, '*detail*'
error
This message is issued in C90 mode when a type specifier is duplicated within a declaration. For example:

```
const const int i = 0;
```

will result in message 29 when in C90 mode, which forbids duplicate specifiers. In C99 and later, duplicate specifiers are ignored and such a construct will instead be greeted with warning 2435.

31 redefinition of symbol *symbol*
error
A data object or function previously defined in this module is being redefined.
Supports MISRA C++ Rule 3-2-1 (Req)
Supports MISRA C++ Rule 3-2-2 (Req)
Supports MISRA C++ Rule 3-2-4 (Req)

32 field size (member *symbol*) should not be zero
error
The length of a field was given as non-positive, (0 or negative).

33 illegal constant '*integer*' in octal constant
error
A constant was badly formed as when an octal constant contains one of the digits `8` or `9`.

34 non-compile-time-constant initializer
error
A non-constant initializer was found for a static data item.

35 initializer has side-effects
error An initializer with side effects was found for a static data item.

40 undeclared identifier *detail*
error Within an expression, an identifier was encountered that had not previously been declared and was not followed by a left parenthesis.

44 need a switch for case-label
error A `case` or `default` statement occurred outside a switch.
Supports MISRA C 2004 Rule 15.1 (Req)
Supports MISRA C 2012 Rule 16.2 (Req)

47 invalid argument type *type* to unary expression
error Unary minus requires an arithmetic operand.

48 indirection requires pointer operand (*type* invalid)
error Unary `*` or the left hand side of the ptr (`->`) operator requires a pointer operand.
Supports MISRA C 2012 Rule 10.1 (Req)

50 attempted to take the address of a non-lvalue of type *type*
error Unary `&` operator requires an lvalue (a value suitable for placement on the left hand side of an assignment operator).

51 expected integral type for bitwise complement operator
error Unary `~` expects an integral type (`signed` or `unsigned char`, `short`, `int`, or `long`).

52 expected an lvalue
error autodecrement (`--`) and autoincrement (`++`) operators require an lvalue (a value suitable for placement on the left hand side of an assignment operator). Remember that casts do not normally produce lvalues. Thus

```
++(char *)p;
```

is illegal according to the ANSI/ISO standard. This construct is allowed by some compilers and is allowed if you use the `+fpc` option (Pointer Casts are lvalues). (See Section 4.10 Flag Options)

53 expected a scalar and not expression of type *type*
error Autodecrement (`--`) and autoincrement (`++`) operators may be applied only to scalars (arithmetics and pointers) or to objects for which these operators have been defined.

54 *division/remainder* by 0
error The constant `0` was used on the right hand side of the division operator (`/`) or the remainder operator (`%`).

56 pointer addition requires an integral type
error Add/subtract operator requires scalar types and pointers may not be added to pointers.

57 **operands to bitwise operator must be integral**
error Bit operators (`&`, `|` and `^`) require integral arguments.

59 **number of bits to shift by must be an integer**
error The amount by which an item can be shifted must be integral.

60 **value to be shifted must be an integer**
error The value to be shifted must be integral.

64 **cannot initialize *string typediff incompatible-types***
error There was a mismatch in types across an assignment. *typediff* specifies the type difference.
Supports MISRA C 2004 Rule 8.4 (Req)
Supports MISRA C 2012 Rule 8.4 (Req)

66 **'void' must be the first and only parameter if specified**
error A `void` type was employed where it is not permitted. If a `void` type is placed in a prototype then it must be the only type within a prototype.

72 **bad option '*option*': *detail***
error Was not able to interpret an option. The option is given in *option*.

76 **can't open file '*file*': *detail***
error *file* is the name of the file. The named file could not be opened for output. *detail* contains information about the failure. This error is issued when PC-lint Plus is directed to write to a file with the options `-oe`/`+oe`, `-os`/`+os`, `-write_file`, or `+stack` but it unable to open the specified file for writing.

82 ***string detail* must not return void expression**
error The ANSI/ISO standard does not allow an expression form of the `return` statement with a `void` function. If you are trying to cast to `void` as in `return (void)f();` and your compiler allows it, suppress this message.

83 **incompatible pointer types (*type* and *type*) with subtraction**
error Two pointers being subtracted have indirect types that differ.

85 **array *symbol* has dimension 0**
error An array (named *symbol*) was declared without a dimension in a context that required a non-zero dimension.

86 **structure *symbol* has zero elements**
error A structure was declared (in a C module) that had no data members. Though legal in C++ this is not legal C.

92 **negative length of *integer* for bit field *integer***
error A negative array dimension or bit field length is not permitted.

95 expected a macro parameter
error
The `#` operator was found within a macro definition but was not immediately followed by a parameter of the macro as is required by the standards.

104 cannot combine with previous '*parameter*' declaration specifier
error
Two consecutive conflicting types were found such as `int` followed by `double`. Remove one of the types.

106 illegal constant
error
A string constant was found within a preprocessor expression as in

```
#if ABC == "abc"
```

Such expressions should be integral expressions.

107 label *name* not defined
error
The *name* appeared in a goto but there was no corresponding label.
Supports MISRA C++ Rule 6-6-2 (Req)

108 invalid context for 'break' statement
error
A `continue` or `break` statement was encountered without an appropriate surrounding context such as a `for`, `while`, or `do` loop or, for the `break` statement only, a surrounding `switch` statement.

111 assignment to const object
error
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.

115 struct/union not defined
error
A reference to a structure or a `union` was made that required a definition and there is no definition in scope. For example, a reference to `p->a` where `p` is a pointer to a `struct` that had not yet been defined in the current module.
Supports MISRA C 2004 Rule 18.1 (Req)

116 inappropriate storage class
error
A storage class other than `register` was given in a section of code that is dedicated to declaring parameters. The section is that part of a function preceding the first left brace.

117 inappropriate storage class
error
A storage class (indicated as either `auto` or `register`) was provided outside any function. Such storage classes are appropriate only within functions.

118 too few arguments (*integer* vs *integer*) for prototype
error
The number of arguments provided for a function was less than the number indicated by a prototype in scope.
Supports MISRA C 2004 Rule 16.6 (Req)

119 **too many arguments (*integer* vs *integer*) for prototype**
error The number of arguments provided for a function was greater than the number indicated by a prototype in scope.
Supports MISRA C 2004 Rule 16.6 (Req)

124 **pointer to void not allowed**
error A pointer to `void` was used in a context that does not permit `void`. This includes subtraction, addition and the relationals (`> >= < <=`).

128 **pointer to function not allowed**
error A pointer to a function was found in an arithmetic context such as subtraction, addition, or one of the relationals (`> >= < <=`).

130 ***type* is not an integral type**
error The expression in a `switch` statement must be some variation of an `int` (possibly `long` or `unsigned`) or an `enum`.

131 **too few arguments provided to function-like macro invocation**
error This message is issued when a macro with arguments (function-like macro) is invoked and an incorrect number of arguments is provided.
Supports MISRA C 2004 Rule 19.8 (Req)

132 **expected function definition**
error A function declaration with identifiers between parentheses is the start of an old-style function definition (K&R style). This is normally followed by optional declarations and a left brace to signal the start of the function body. Either replace the identifier(s) with type(s) or complete the function with a function body.

136 **illegal macro name**
error The ANSI/ISO standard restricts the use of certain names as macros. `defined` is on the restricted list.
Supports MISRA C 2012 Rule 21.1 (Req)

138 **cannot create recursive relationship between '*strong-type*' and '*strong-type*'**
error An attempt was made to add a strong type `parent` to a `typedef` type. The attempt is either explicit (with the `-strong` option) or implicit with the use of a `typedef` to a known strong type. This attempt would have caused a loop in the strong parent relationship. Such loops are simply not tolerated.

139 **cannot take sizeof a function**
error There is an attempt to take the `sizeof` a function.

148 **member *name* previously declared**
error The indicated member was previously declared within the same structure or union. Although a redeclaration of a function may appear benign it is just not permitted by the rules of the language. One of the declarations should be removed.

157 no data may follow an incomplete array
error
An incomplete array is allowed within a struct of a C99 or C++ program but no data is allowed to appear after this array. For example

```
{struct A { int x; int a[]; int b; };
```

This diagnostic is issued when the 'b' is seen.

160 the sequence ({ is non standard and is taken to introduce a GNU statement expression
error
PC-lint Plus encountered the sequence '({' in a context where an expression (possibly a sub-expression) is expected. For example:

```
int n = ({   //Error 160 here
      int y = foo ();
      int z;
      if (y > 0)
          z = y;
      else z = - y;
      z; })
      // Now n has the last value of z.
```

In addition to being a non-standard GNU extension, there are some caveats described in the GCC documentation (especially when used in C++) that can lead to subtle bugs. Programmers who intend to work only with C code with the GNU extensions may safely disable this diagnostic.

161 repeated use of parameter *symbol* in parameter list
error
The name of a function parameter was repeated. For example:

```
void f( int n, int m, int n ) {}
```

will cause this message to be issued. Names of parameters for a given function must all be different.

175 cannot pass *string* to variadic *string*; expected type from format string was *type*
error
An initializer list or an expression of a type that cannot be passed as a variadic function argument was given as the argument to a `printf`/`scanf` style function. The *string* parameter specifies the type of the argument passed, the *type* parameter specifies the type that was expected from the format string.

176 operand of type *type* cannot be cast to function pointer type *type*
error
An attempt was made to perform an illegal cast from a type (such as a float) to a function pointer for which such conversion is undefined.
Supports MISRA C 2004 Rule 11.1 (Req)

177 operand of type *type* cannot be cast to object pointer type *type*
error
An attempt was made to perform an illegal cast from a type (such as a float) to an object pointer for which such conversion is undefined.
Supports MISRA C 2004 Rule 11.2 (Req)
Supports MISRA C 2012 Rule 11.7 (Req)

178 function pointer of type *type* cannot be cast to type *type*
error
An attempt was made to perform an illegal cast from a function pointer to a type (such as a float) for which such conversion is undefined.
Supports MISRA C 2004 Rule 11.1 (Req)

179 **object pointer of type *type* cannot be cast to type *type***
error An attempt was made to perform an illegal cast from an object pointer to a type (such as a float) for which such conversion is undefined.
Supports MISRA C 2004 Rule 11.2 (Req)
Supports MISRA C 2012 Rule 11.7 (Req)

305 **unable to open module '*file*'**
error *file* is the name of the module. The named module could not be opened for reading. Perhaps you misspelled the name.

307 **cannot open indirect file '*file*'**
error *file* is the name of the indirect file. The named indirect file (probably ending in `.lnt`) could not be opened for reading.

308 **can't write to file '*file*' for PCH construction**
error `stdout` was found to equal NULL. This is most unusual.

309 **#error *detail***
error The `#error` directive was encountered. This error is fatal by default, but can be bypassed using the `fce` flag.

314 **cannot use indirect file '*file*' again**
error The indirect file named was previously encountered. If this was not an accident, you may suppress this message.

315 **message limit exceeded**
error The maximum number of messages specified using the `-limit` option was exceeded.

318 **EOF for a module found within a macro argument list**
error We found the end of a module within the argument list of a macro. Since such situations are almost certain to be erroneous, we gracefully shut down, alerting the User to the reason.

319 **size option misconfiguration: '*type*' has size *integer* and '*type*' has size *integer***
error A fatal inconsistency in the sizes of the fundamental data types was introduced by use of the size options. The `-s` option allows for configuration of the sizes of the fundamental data types. If these options are used to specify sizes that violate the basic tenets of the language, this message will be issued. Such an example would include specifying a byte size for `short int` that is larger than `int`.

322 **unable to open include file '*file*'**
error *file* is the name of the include file, which could not be opened. Directory search is controlled by options: `-i`, `+fdi`, `+fsi` and the `INCLUDE` environment variable (See Section 12.2.1 INCLUDE Environment Variable). This is a suppressible fatal message. If option `-e322` is used processing will continue.

330 static_assert failed *string*
error
This message is issued when the constant-expression of a static-assert-declaration (either C11's `_Static_assert` or C++11's `static_assert`) evaluates to false. If PC-lint Plus issues this error message but the compiler does not, see whether the Lint configuration matches the compiler configuration: consider potential differences in pre-defined macros and include search options, and ensure that size options match the target machine. Differences in these configuration details could lead to differences in evaluation of the constant-expression.

331 file '*parameter*' has been modified since the precompiled header '*parameter*' was built
error
Use of the specified precompiled header was requested but was found to contain a reference to a header file that has been updated since the precompiled header was built. Since the precompiled header may no longer accurately represent the state of the corresponding header file, PC-lint Plus will terminate. To resolve the issue either rebuild the precompiled header file or remove the option to use the pre-compiled header.

333 cannot open '*file*' for *string* in secure mode
error
A 'forbidden' file was opened. Opening such a file is considered a security violation by a hosted implementation.

334 precompiled header failure: '*string*'; skipping this module; consider deleting the PCH, '*string*', and trying again
error
This message is given when the precompiled header file is missing, older than the original file, created by a previous incompatible version of PC-lint Plus, created using a different target configuration, or another issue that prevents the file from being loaded. The PCH file will be skipped. If this error is encountered, the PCH file should be deleted and reconstituted using the current version of PC-lint Plus with the same options that will be used when loading the file.

336 source file is not valid UTF-8
error
Source files are expected to be encoded as ASCII text, UTF-8 text, or UTF-16 text. The provided source file was presumed to contain UTF-8 text but an invalid byte sequence was encountered.

338 precompiled header error: *string*; skipping this module; consider examining include path options and trying again
error
This error indicates that there was an inconsistency in the way the precompiled header file was created and the way it is being used that prevents it from being loaded. The details of the issue are provided in the message text.

339 precompiled header for '*string*' was not created due to errors
error
Precompiled header file creation was requested but an error occurred during the processing of a precompiled header candidate file that rendered the corresponding AST information unsuitable for use in a precompiled header file.

365 command pipe error: *string*
error
An error has occurred while processing a request related to a command pipe program. The details of the error are specified in the message in '*string*'.

366 regex error: *string*
error
An invalid regular expression has been used with the `-cond` option. The specific error is provided in '*string*'.

367 **maximum hook recursion depth (*integer* levels) reached**
error A limit has been reached on the number of recursively executing hooks. The limit that was reached is specified by '*integer*'.

368 **invalid conditional expression: *string***
error An invalid conditional expression has been provided to the `-cond` option. The specific error is provided in the text of the message.

369 **hook field error while processing '*string*': *string***
error An invalid field name was provided in a hook field specifier or an AST walk action was attempted on a non-walkable hook field.

370 **options executed within a module cannot invoke additional modules**
error An attempt was made to process a new module from within the module being processed. For example, a lint comment might contain an `-indirect` option resulting in the processing of options from a `.lnt` file. If this indirect file contains the name of a module to process, the module will not be opened and this message will be issued instead. Processing will then continue normally for the current module.

373 **lint comments cannot appear after a #include directive on the same line**
error A lint comment appeared on the same line as an `#include` directive. Such usage is not currently supported and the lint comment will be ignored. Either place the lint comment before the `#include` directive, on the next line, or inside the file being included.

401 **symbol *symbol* not previously declared static**
warning The indicated *symbol* declared `static` was previously declared without the `static` storage class. This is technically a violation of the ANSI/ISO standard. Some compilers will accept this situation without complaint and regard the *symbol* as `static`.

Supports MISRA C 2004 Rule 8.11 (Req)

Supports MISRA C++ Rule 3-3-2 (Req)

402 **static *function/variable symbol* not defined**
warning The named *symbol* was declared as a `static` function in the current module and was referenced but was not defined (in the module).

404 **definition of *type* starts in '*file*' but ends in '*file*'**
warning A `struct` (or `union` or `enum`) definition was started within a header file but was not completed within the same header file.

407 **inconsistent use of tag *symbol***
warning A tag specified as a `union`, `struct` or `enum` was respecified as being one of the other two in the same module. For example:

```
struct tag *p;
union tag *q;
```

will elicit this message.

408 **case expression type (*type*) differs from switch expression type (*type*)**
warning The expression within a `case` does not agree exactly with the type within the `switch` expression. For example, an enumerated type is matched against an `int`.

409 **integer base for subscript operator is suspicious**
warning An expression of the form *i*`[...]` was encountered where *i* is an integral expression. This could be legitimate depending on the subscript operand. For example, if `i` is an `int` and `a` is an array then `i[a]` is legitimate but unusual. If this is your coding style, suppress this message.

410 **size_t not what was expected from fzl and/or fzu, using *type***
warning This warning is issued if you had previously attempted to set the type of `sizeof` by use of the options `+fzl`, `-fzl`, or `-fzu`, and a later `size_t` declaration contradicts the setting. This usually means you are attempting to lint programs for another system using header files for your own system. If this is the case we suggest you create a directory housing header files for that foreign system, alter `size_t` within that directory, and lint using that directory.

411 **ptrdiff_t not what was expected from fdl option, using *type***
warning This warning is issued if you had previously attempted to set the type of pointer differences by use of the `fdl` option and a later `ptrdiff_t` declaration contradicts the setting. See suggestion in Error Message `410`.

413 **likely use of null pointer *symbol***
warning From information gleaned from earlier statements, it appears likely that a null pointer (a pointer whose value is 0) has been used in a context where null pointers are inappropriate. Information leading to this determination is provided as a series of supplemental messages. See also message `613`.

414 **possible division by zero**
warning The second argument to either the division operator (`/`) or the modulus operator (`%`) may be zero. Information is taken from earlier statements including assignments, initialization and tests. See Chapter 8 Value Tracking.

415 **likely out of bounds pointer access: excess of *integer* byte(s)**
warning An out-of-bounds pointer was likely accessed. The parameter *integer* gives some idea how far out of bounds the pointer may be, measured in bytes. For example:

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

results in a message containing the phrase 'excess of 4 bytes' if the size of `int` is 4. See Chapter 8 Value Tracking.
Supports MISRA C 2012 Rule 18.1 (Req)
Supports MISRA C++ Rule 5-0-16 (Req)

416 **likely creating out-of-bounds pointer: excess of *integer* byte(s)**
warning An out-of-bounds pointer was created. See message `415` for a description of the *integer* parameter. *integer* and *string*. For example:

```
int a[10];
   ...
f( a + 11 );
```

Here, an illicit pointer value is created and is flagged as such by PC-lint Plus. Note that the pointer `a+10` is not considered by PC-lint Plus to be the creation of an out-of-bounds pointer. This is because ANSI/ISO C explicitly allows pointing just beyond an array. Access through `a+10`, however, as in `*(a+10)` or the more familiar `a[10]`, would be considered erroneous but in that case message **415** would be issued. See Chapter 8 Value Tracking.
Supports MISRA C 2012 Rule 18.1 (Req)
Supports MISRA C++ Rule 5-0-16 (Req)

417 integral constant '*string*' has precision *integer* which is longer than long long int
warning
The longest possible integer is by default 8 bytes (see the `+fll` flag and then the `-sll#` option). An integral constant was found to be even larger than such a quantity. For example: `0xFFFF0000FFFF0000F` requires 68 bits and would by default elicit this message. *string* is the token in error, and *integer* is the binary precision.

418 passing null pointer to function *symbol*, *context*
warning
A NULL pointer is being passed to a function identified by *symbol*. The argument in question is given by *context*. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option `-function` or `-sem`. See Section 9.1 Function Mimicry (-function) and Section 9.2.1 Possible Semantics.
Supports MISRA C 2012 Directive 4.11 (Req)

419 apparent data overrun for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)
warning
This message is for data transfer functions such as `memcpy`, `strcpy`, `fgets`, etc. when the size indicated by the first cited argument (or arguments) exceeds the size of the buffer area cited by the second. The message may also be issued for user functions via the `-function` option. See Section 9.1 Function Mimicry (-function) and Section 9.2.1 Possible Semantics.
Supports MISRA C 2012 Directive 4.11 (Req)

420 apparent access beyond array for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)
warning
This message is issued for several library functions (such as `fwrite`, `memcmp`, etc.) wherein there is an apparent attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified. The function is specified by *symbol* and the arguments are identified by argument number. See Section 9.1 Function Mimicry (-function) and Section 9.2.1 Possible Semantics.
Supports MISRA C 2012 Directive 4.11 (Req)

421 caution – function *symbol* is considered dangerous
warning
This message is issued (by default) for the built-in function `gets`. This function is considered dangerous because there is no mechanism to ensure that the buffer provided as first argument will not overflow. Numerous exploits and vulnerabilities are attributed to the `gets` function including the Morris worm, which exploited the use of the `gets` function in the fingered program of target machines. Through the `-function` option or the `dangerous` semantic (9.2.1 dangerous), the user may designate other functions as dangerous. See also `-deprecate`.

422 passing to function *symbol* a negative value (*integer*) *context*
warning
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.
Supports MISRA C 2012 Directive 4.11 (Req)

423 **warning** **assignment to custodial pointer *symbol* likely creates memory leak**
An assignment was made to a pointer variable (designated by *symbol*), which appeared to already be holding the address of an allocated object that had not been freed. The allocation of memory that is not freed is considered a memory leak.

424 **warning** ***string* is not appropriate for deallocating *string***
This message indicates that a deallocation (`free`, `delete`, or `delete[]`) as specified by the first *string* parameter is inappropriate for the data being freed. [4, Item 5]

The kind of data (specified by the second *string* parameter) is one or more of: `malloc`, `new`, `new[]`, `static`, `auto`, `member`, `modified` or `constant`. These have the meanings as described below:

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

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

  `p+1` is considered modified.
- `constant` data is the result of casting a constant to a pointer. E.g.

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

See also message 673.
Supports MISRA C 2012 Rule 22.2 (Mand)

425 **warning** **'*message*' in processing semantic '*string*' at token '*token*'**
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 **warning** **call to function *symbol* violates semantic '*string*'**
This warning message is issued when a user semantic (as defined by -sem) is violated. '*string*' is the subportion of the semantic that was violated. For example:

```
//lint -sem( f, 1n > 10 && 2n > 10 )
void f( int, int );
...
    f( 2, 20 );
```

results in the message:

```
Call to function 'f(int, int)' violates semantic '(1n>10)'
```

427 **// comment continued via back-slash**
warning
The line that starts a C++ style comment ends with a back-slash causing the next line to be absorbed into the comment, which may not be the intended behavior. If you really intend the next line to be a comment, the line should be started with its own double slash (//) or the entire region replaced with a block comment.
Supports MISRA C 2012 Rule 3.2 (Req)

428 **likely indexing before the beginning of an allocation**
warning
A negative integer was added to an array or to a pointer to an allocated area (allocated by `malloc`, `operator new`, etc.) This message is not given for pointers whose origin is unknown since a negative subscript is, in general, legal.

The addition could have occurred as part of a subscript operation or as part of a pointer arithmetic operation.
Supports MISRA C 2012 Rule 18.1 (Req)

429 **custodial pointer *symbol* likely not freed nor returned**
warning
A pointer of `auto` storage class was allocated storage, which was neither freed nor returned to the caller. This represents a "memory leak". A pointer is considered custodial if it uniquely points to the storage area. It is not considered custodial if it has been copied. Thus:

```
int *p = new int[20];   // p is a custodial pointer
int *q = p;             // p is no longer custodial
p = new int[20];        // p again becomes custodial
q = p + 0;              // p remains custodial
```

Here `p` does not lose its custodial property by merely participating in an arithmetic operation.

A pointer can lose its custodial property by passing the pointer to a function. If the parameter of the function is typed pointer to `const` or if the function is a library function, that assumption is not made. For example

```
p = malloc(10);
strcpy (p, "hello");
```

Then `p` still has custody of storage allocated.

It is possible to indicate via semantic options that a function will take custody of a pointer. See 9.2.1 custodial(i). It is possible to declare that no functions take custody other than those specified in a `-sem` option. See also Flag `ffc` (Functions take custody).
Supports MISRA C 2012 Rule 22.1 (Req)

430 **use of '@' is non-standard**
warning
Many compilers for embedded systems have a declaration syntax that specifies a location in place of an initial value for a variable. For example:

```
int x @0x2000;
```

specifies that variable `x` is actually location `0x2000`. This message is a reminder that this syntax is non-standard (although quite common). If you are using this syntax on purpose, suppress this message.

432 suspicious argument to dynamic allocation function
warning
The following pattern was detected:

```
malloc( strlen(e+1) )
```

where *e* is some expression. This is suspicious because it closely resembles the commonly used pattern:

```
malloc( strlen(e)+1 )
```

If you really intended to use the first pattern then an equivalent expression that will not raise this error is:

```
malloc( strlen(e)-1 )
```

433 allocated area not large enough for pointer (*integer* vs *string*)
warning
An allocation was assigned to a pointer whose reach extends beyond the area that was allocated. This would usually happen only with library allocation routines such as `malloc` and `calloc`. For example:

```
int *p = malloc(1);
```

This message is also provided for user-declared allocation functions. For example, if a user's own allocation function is provided with the following semantic:

```
-sem(ouralloc,@P==malloc(1n))
```

We would report the same message. Please note that it is necessary to designate that the returned area is freshly allocated (ala `malloc`).

This message is always given in conjunction with the more general Informational Message 826.

434 white space ignored between back-slash and new-line
warning
According to the C and C++ standards, any back-slash followed immediately by a new-line results in the deletion of both characters. For example:

```
#define A \
        34
```

defines `A` to be `34`. If a blank or tab intervenes between the back-slash and the new-line then according to a strict interpretation of the standard you have defined `A` to be a back-slash followed by blank or tab. But this blank is invisible to the naked eye and hence could lead to confusion. Worse, some compilers silently ignore the white-space and the program becomes non-portable.

You should never deliberately place a blank at the end of a line and any such blanks should be removed. If you really need to define a macro to be back-slash blank you can use a comment as in:

```
#define A \  /* commentary */
```

435 integral constant '*string*' has precision *integer*, use +fll to enable long long
warning
An integer constant was found that had a precision that was too large for a `long` but would fit within a `long long`. Yet the `+fll` flag that enables the `long long` type was not set.

Check the sizes that you specified for long (-sl#) and for long long (-sll#) and make sure they are correct. Turn on +fll if your compiler supports long long. Otherwise use smaller constants.

436 warning **preprocessor directive in invocation of macro**
A function like macro was invoked whose arguments extended for multiple lines, which included preprocessor statements. This is almost certainly an error brought about by a missing right parenthesis.

By the rules of the C and C++ standards, the result of this behavior is undefined. For this reason some compilers treat the apparent preprocessor directive as a directive. However, avoiding this construct is recommended for portability.

Supports MISRA C 2004 Rule 19.9 (Req)
Supports MISRA C 2012 Rule 20.6 (Req)
Supports MISRA C++ Rule 16-0-5 (Req)

437 warning **passing a *class/struct* to an elliptic argument**
A `struct` or `class` is being passed to a function at a parameter position identified by an ellipsis. For example:

```
void g()
    {
    struct A { int a; } x;
    void f( int, ... );
    f( 1, x );
    ...
    }
```

This is sufficiently unusual that it is worth pointing out in the likelihood that this is unintended. The situation becomes more severe in the case of a non-POD `struct [10]`. In this case the behavior is considered undefined.

438 warning **last value assigned to *symbol* not used**
A value had been assigned to a variable that was not subsequently used. The message is issued either at a return statement or at the end of a block when the variable goes out of scope. For example, consider the following function:

```
void f( int n )
    {
    int x = 0, y = 1;
    if( n > 0 )
        {
        int z;
        z = x + y;
        if( n > z ) { x = 3; return; }
        z = 12;
        }
    }
```

Here we can report that `x` was assigned a value that had not been used by the time the return statement had been encountered. We also report that the most recently assigned value to `z` is unused at the point that `z` goes out of scope. See also Informational message 838 and flags -fiw and -fiz.

Supports MISRA C 2012 Rule 2.2 (Req)
Supports MISRA C++ Rule 0-1-6 (Req)
Supports MISRA C++ Rule 0-1-9 (Req)

440 **for statement condition variable is inconsistent with modification variable**
warning A `for` clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd `for` clause expression, does not match the variable that is tested in the 2nd `for` clause expression. For example:

```
for( i = 0; i < 10; j++ )
    ...
```

would draw this complaint since the 'i' of the 2nd expression does not match the 'j' of the third expression.
Supports MISRA C 2004 Rule 13.5 (Req)

442 **for statement condition and increment directions are inconsistent**
warning A `for` clause was encountered that appeared to have a parity problem. For example:

```
for( i = 0; i < 10; i-- )
    ...
```

Here the test for `i` less than 10 seems inconsistent with the 3rd expression of the `for` clause, which decreases the value of `i`. This same message would be given if `i` were being increased by the 3rd expression and was being tested for being greater than some value in the 2nd expression.

443 **for statement initializer variable is inconsistent with modification variable**
warning A `for` clause has a suspicious structure. The loop variable, as determined by an examination of the 3rd `for` clause expression, does not match the variable that is initialized in the 1st expression. For example:

```
for( ii = 0; i < 10; i++ )
    ...
```

would draw this complaint since the 'ii' of the 1st expression does not match the 'i' of the third expression.
Supports MISRA C 2004 Rule 13.5 (Req)

444 **for statement condition tests *incremented/decremented* pointer for null**
warning The following kind of situation has been detected:

```
for( ... ; p == NULL; p++ )
     ...
```

A loop variable being incremented or decremented would not normally be checked to see if it is NULL. This is more likely a programmer error.

445 **reuse of for loop variable *symbol***
warning A `for` loop nested within another `for` loop employed the same loop variable. For example:

```
for( i = 0; i < 100; i++ ) {
    for( i = 0; i < n; i++ ) { ... }
}
```

446 **side-effect in initializer list**
warning An initializer containing a side effect can be potentially troublesome. For example, the code:

```
void f( int i ) {
    int a[2] = {i++, i++};
 }
```

The values of the array elements are unspecified because the order of evaluation is unspecified by the C standard.
Supports MISRA C 2012 Rule 13.1 (Req)

447 **extraneous whitespace found in include directive for file *file*; Opening file *file***
warning A named file was found to contain either leading or trailing whitespace in the `#include` directive. While legal, the ISO Standards allow compilers to define how files are specified or the header is identified, including the appearance of whitespace characters immediately after the `<` or opening `"` or before the `>` or closing `"`. Since filenames tend not to contain leading or trailing whitespace, PC-lint Plus ignores the (apparently) extraneous characters and processes the directive as though the characters were never given. The use of a `-efile` option on either *file* for this message will cause Lint to process `#include`'s with whitespace intact.

448 **possible access *integer* bytes beyond null terminator by '*operator*'**
warning Accessing past the terminating nul character is often an indication of a programmer error. For example:

```
char buf[20];
strcpy( buf, "a" );
char c = buf[4]; // legal but suspect
```

Although `buf` has 20 characters, after the `strcpy`, there would be only two that the programmer would normally be interested in.

449 **memory was likely previously deallocated**
warning A pointer variable (designated in the message) was freed or deleted in an earlier statement.
Supports MISRA C 2012 Rule 22.2 (Mand)
Supports MISRA C 2012 Rule 22.6 (Mand)

450 **namespace *symbol* declared within an extern "C" region**
warning A namespace was declared either with an `extern "C"` specifier or within an `extern "C"` region. The ISO C++ standard leaves the effects of such unspecified. If an `extern "C"` specification is necessary for the declarations within the namespace, it should be inside the namespace rather than outside.

451 **header file '*string*' repeatedly included but has no header guard**
warning The file named in the message has already been included in the current module. Moreover it has been determined that this header does not have a standard include guard. A standard include guard has the form

```
#ifndef Name
#define Name
...
#endif
```

or

```
#if !defined(X)
#define X
...
#endif
```

with nothing but comments before and after this sequence and nothing but comments between the `#if`/`#ifndef` and the `#define` *name*.

This warning may also be accompanied by a 537 (repeated include header). Message 537 is often suppressed because if you are working with include guards it is not a helpful message. However, the message `451` should be left on in order to check the consistency of the include guards themselves.

This message is not issued for headers that employ `#pragma once`
Supports MISRA C 2004 Rule 19.15 (Req)
Supports MISRA C 2012 Directive 4.10 (Req)

452 ***type* redefinition with different types *type***
warning A `typedef` symbol is being declared to be a different type. This can be legal, especially with multiple modules, but is not good programming practice. It interferes with program legibility.

453 **function *symbol*, previously designated pure, *reason symbol***
warning A semantic option designated that the named function, *symbol*, is pure (lacking non-local side-effects): see the `pure` semantic in Chapter 9.1.2 Semantics. However, an impurity was detected. Such impurities include calling a function through a function pointer, accessing a volatile variable, modifying a static variable or calling a function whose purity PC-lint Plus cannot verify. *Reason* describes which of these reasons apply and the second *symbol* parameter shows the related variable or function as appropriate.

Despite the inconsistency reported, the function will continue to be regarded as pure.

463 **could not parse '*string*' as a strong type: *string***
warning This message is issued when a parse failure occurs when parsing a type specified with the `-strong` option. The first *string* contains the type specification that caused the error and the second *string* provides additional information about the error such as "unmatched right parenthesis".

464 **buffer argument will be copied into itself**
warning This is issued when we encounter a function argument expression used in such a way that there will be an attempt to copy its contents onto itself. E.g.

```
sprintf( s, "%s", s );
```

466 **conversion *to/from* pointer to function with no prototype (*context*)**
warning A pointer to a function without a prototype was assigned to or from another pointer to function. While assigning a pointer to function with a prototype, to one without a prototype is legal in ISO C, unexpected behavior may occur too easily. For example:

```
char *(*pf)();
char *strchr(const char *);
void g()
    {
    pf = strchr; // Msg 466
    pf(12, 2);   // undefined behavior
}
```

473 **argument '*string*' is of insufficient length for array parameter *symbol* declared as *type***
warning This message is issued when a function declared with a constant-sized array parameter is passed an argument which can be determined, using Value Tracking, to either be null or to point to an area that is smaller than

the size of the array. For example:

```
void init(unsigned char array[10]);

void *malloc(unsigned);

void foo() {
    unsigned char array1[5];
    unsigned int array2[3];
    unsigned char *pc1 = malloc(8);
    unsigned char *pc2 = (unsigned char *)array2;

    init(array1);    // 473 - array1 is 5 bytes, init expects 10
    init(pc1);       // 473 - pc1 points to 8 bytes (or is null)
    init(pc2);       // Okay - assuming ints are 4 bytes or larger
    init(0);         // 473 - null argument
}
```

Supports MISRA C 2012 Rule 17.5 (Adv)

474 warning **constant switch condition '*string*' not handled by switch**
The condition of a `switch` was a constant expression, e.g. `switch(7)`. Furthermore, there is no default case and no case statement that matches the constant expression. '*string*' contains the constant used as the `switch` condition.

477 warning **array *symbol* could be declared static**
An array of `const` qualified objects was defined at function scope without static storage duration. Repeatedly reallocating such arrays can impair performance.

483 warning **switching on a boolean value**
At least one standards organization has expressed the perspective, if the expression of a `switch` statement is boolean in nature, `if-else` should be used instead.
Supports MISRA C 2012 Rule 16.7 (Req)
Supports MISRA C++ Rule 6-4-7 (Req)

484 warning **stringize operator followed by macro parameter followed by pasting operator**
Due to order of evaluation issues, the mixing of stringizing and pasting operators, particularly when appearing in the order `#` *parameter* `##`, results in unspecified behavior.
Supports MISRA C 2012 Rule 20.11 (Req)

485 warning **duplicate initialization of object element**
In addition to the behavior being unspecified when the use of designated initializers results in duplicate object initialization, assigning to an array element or structure member more than once in an initializer is typically a logic error.
Supports MISRA C 2012 Rule 9.4 (Req)

486 warning **writing to file opened as read-only**
A file pointer was obtained with a call to an `fopen`-like function in read-only mode. This pointer was then

used in an attempt to write to the file. The ISO standards leave the behavior in such cases as unspecified.

488 **enumerator *symbol* reuses the constant value '*integer*' previously used implicitly by enumerator *symbol***
warning
Two enumerators have the same value and at least one received that value implicitly. For example:

```
enum colors { red, blue, green = 1 };
```

will elicit this informational message while

```
enum colors { red, blue = 1, green = 1 };
```

will not.
Supports MISRA C 2012 Rule 8.12 (Req)

489 **attempting to modify the contents of a string literal**
warning
An assignment to an element of a string literal was seen. Doing so results in undefined behavior.
Supports MISRA C 2012 Rule 7.4 (Req)

490 ***string***
warning
This message is issued as a result of processing a `#warning` preprocessor directive. *string* is the message provided to the directive.

491 **non-standard use of 'defined' preprocessor operator: *detail***
warning
The ISO standards restrict the use of the `defined` preprocessor keyword to either

```
defined(identifier)
defined identifier
```

Additionally, the preprocessor operator may not result from the expansion of another macro. This diagnostic highlights departures from these requirements as non-portable code.
Supports MISRA C 2004 Rule 19.14 (Req)
Supports MISRA C++ Rule 16-1-1 (Req)

492 **incomplete format specifier '*string*'**
warning
A format specifier for a `printf/scanf` style function was started but did not contain a conversion specifier. For example:

```
printf("%ll", 3LL);
```

will yield the message:

```
incomplete format specifier '%ll'
```

493 **position arguments in format strings start counting at 1 (not 0)**
warning
A format specifier for a `printf/scanf` style function attempted to reference the argument at position 0 but positional arguments are indexed at 1 so this is not valid. For example:

```
printf("%0$d", 3);
```

494 **data argument position '*integer*' exceeds the number of data arguments (*integer*)**
warning
A format specifier for a `printf/scanf` style function utilizing positional arguments contained a reference to a non-existent argument, which results in undefined behavior. For example:

```
printf("%2$d", j)
```

will yield the message:

```
data argument position '2' exceeds the number of data arguments (1)
```

495 format string body contains NUL character
warning A format string for a `printf`/`scanf` style function contains a nul character in the body of the string. The receiving function will not be able to access the portion of the string after this character and its inclusion is likely a mistake. For example:

```
printf("%d\0%d", 1, 2);
```

will elicit this message.

496 format string is not null terminated
warning The format string provided to a `printf`/`scanf` style function is not terminated with a nul character, which will cause the function to read past the end of the string causing undefined behavior.

497 format string is empty
warning An empty format string was provided to a `printf` or `scanf` like function. Calling these functions with an empty format string is legal but suspect as there is no effect to doing so.

498 unbounded scanf conversion specifier '*string*' may result in buffer overflow
warning A `%s` or `%[` conversion specifier was encountered in the format string of a `scanf`-like function that did not contain a maximum field width. Since the `%s` and `%[` conversion specifiers read characters into the target buffer until either the maximum field width is reached or a prescribed character is encountered, failing to provide a maximum field width can easily result in buffer overflow. '*string*' contains the unbounded format specifier.

499 using length modifier '*string*' with conversion specifier '*string*' is not supported by ISO C
warning Within the format for a `printf` or `scanf` like function, a length modifier was combined with a conversion specifier that is not supported by Standard C.

501 negation of value of unsigned type *type* yields a value of unsigned type *type*
warning The unary minus operator was applied to an unsigned type. The resulting value is a positive unsigned quantity and may not be what was intended.

502 applying bitwise not to signed quantity
warning Unary `~` being a bit operator would more logically be applied to unsigned quantities rather than signed quantities.

503 boolean argument to relational
warning Normally a relational would not have a Boolean as argument. An example of this is `a < b < c`, which is technically legal but does not produce the same result as the mathematical expression, which it resembles.

504 unusual shift operation (*string*)
warning
Either the quantity being shifted or the amount by which a quantity is to be shifted was derived in an unusual way such as with a bit-wise logical operator, a negation, or with an unparenthesized expression. If the shift value is a compound expression that is not parenthesized, parenthesize it.

505 redundant left argument to comma
warning
The left argument to the comma operator had no side effects in its top-most operator and hence is redundant.
Supports MISRA C 2004 Rule 14.2 (Req)
Supports MISRA C 2012 Rule 2.2 (Req)

506 integer constant expression used in boolean context
warning
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.
Supports MISRA C 2004 Rule 13.7 (Req)
Supports MISRA C 2012 Rule 2.1 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)

507 explicit cast from *type* to *type* (*integer* bits to *integer* bits)
warning
A cast was made to an integral quantity from a pointer and according to other information given or implied it would not fit. For example, a cast to an `unsigned int` was specified and information provided by the options indicate that a pointer is larger than an `int`. Two *integers* are supplied. The first is the size in bytes of the pointer and the second is the size in bytes of the integer.

511 explicit cast from *type* to *type* (*integer* bits to *integer* bits)
warning
A cast was made from an integral type to a pointer and the size of the quantity was too large to fit into the pointer. For example if a `long` is cast to a pointer and if options indicate that a `long` is larger than a pointer, this warning would be reported.

513 the option '*string*' is not currently supported
warning
The specified option is not supported in this version of PC-lint Plus but may be available in a future version.

514 boolean argument to *arithmetic/bitwise* operator '*operator*'
warning
An argument to an arithmetic operator (`+ - / * %`) or a bit-wise logical operator (`| & ^`) was a Boolean. This can often happen by accident as in:

```
if( flags \& 4 == 0 )
```

where the `==`, having higher precedence than `&`, is done first (to the puzzlement of the programmer).

517 defined not K&R
warning
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 parenthesis around type name in *context* expression
warning
`sizeof` *type* is not strict C. `sizeof(`*type*`)` or `sizeof` *expression* are both permissible.

519 **explicit cast from *type* to *type* (*integer* bits to *integer* bits)**
warning An attempt was made to cast a pointer to a pointer of unequal size. This could occur for example in a P model where pointers to functions require 4 bytes whereas pointers to data require only 2. This error message can be circumvented by first casting the pointer to an integral quantity (`int` or `long`) before casting to a pointer.

520 **first clause of for statement lacks side effects**
warning The first expression of a `for` clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning 522.
Supports MISRA C 2012 Rule 2.2 (Req)

521 **third clause of for statement lacks side effects**
warning The third expression of a `for` clause should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). See Warning 522.
Supports MISRA C 2012 Rule 2.2 (Req)

522 **highest operation, *string* '*name*', lacks side effects**
warning If a statement consists only of an expression, it should either be one of the privileged operators: assignment, increment, decrement or a call to an impure function or one modifying its argument(s). For example, if operator `*` is the built-in operator, the statement `*p++;` draws this message with *string* equal to operator and *name* equal to `*`. But note that `p++;` does not. This is because the highest operator in the former case is '`*`', which has no side effects whereas `p++` does. It is possible for a function to have no side-effects. Such a function is called pure. See the discussion of the pure semantic in Section 9.2.1 Possible Semantics. For example:

```
void f() { int n = 3; n++; }
void g() { f(); }
```

will trigger this message with *string* in the message equal to function and *name* equal to `f`.

The definition of pure and impure functions and function calls that have side effects are given in the discussion of the pure semantic in Chapter 9.1.2 Semantics.
Supports MISRA C 2004 Rule 14.2 (Req)
Supports MISRA C 2012 Rule 2.2 (Req)

523 **expression statement involving *string* '*name*' lacks side effects**
warning This message is similar to 522 but is issued only if the entire statement lacks side effects. For example:

```
void foo() {
    int i = 0;
    i++ + 1;
}
```

While the operator `+` lacks side effects, the statement doesn't so 522 will be issued here but not 523.

524 **implicit truncation from *type* to *type***
warning There is a possible loss of a fraction in converting from a float to an integral quantity. Use of a cast will suppress this message.

525 **unexpected negative indentation**
warning The current line was found to be negatively indented (i.e., not indented as much) from the indicated line.

The latter corresponds to a clause introducing a control structure, and statements and other control clauses and braces within its scope are expected to have no less indentation. If tabs within your program are other than 8 blanks you should use the `-t` option (See Section 11.4 Indentation Checking).

526 symbol *symbol* is not defined
warning
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 (`-unit_check` option). Please note that a declaration, even one bearing prototype information is not a definition. See the glossary at the beginning of this chapter. If the *symbol* is a library symbol, make sure that it is declared in a header file that you're including. Also make sure that the header file is regarded by PC-lint Plus as a Library Header file. Alternatively, the symbol may be declared in a Library Module. See Section 5.1 Library Header Files and Section 5.2 Library Modules for a further discussion.

527 statement is unreachable due to unconditional transfer of control by '*string*' statement
warning
A portion of the program cannot be reached. The control mechanism responsible for unconditionally diverting flow away from the specified area is given by *string*.
Supports MISRA C 2004 Rule 14.1 (Req)
Supports MISRA C 2012 Rule 2.1 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)

528 static symbol *symbol* not referenced
warning
The named static variable or static function was not referenced in the module after having been declared.
Supports MISRA C++ Rule 0-1-3 (Req)
Supports MISRA C++ Rule 0-1-4 (Req)
Supports MISRA C++ Rule 0-1-10 (Req)

529 local variable *symbol* declared in *symbol* not subsequently referenced
warning
The named variable was declared but not referenced in a function.
Supports MISRA C++ Rule 0-1-3 (Req)
Supports MISRA C++ Rule 0-1-4 (Req)

530 likely using an uninitialized value
warning
An `auto` variable was used before it was initialized.
Supports MISRA C 2004 Rule 9.1 (Req)
Supports MISRA C 2012 Rule 9.1 (Mand)
Supports MISRA C++ Rule 8-5-1 (Req)

531 width of anonymous bit-field (*integer* bits) exceeds *string* of its type (*integer* bit(s))
warning
The size given for a bit field of a structure exceeds the size of an `int`.

533 function *symbol* should return a value
warning
A function declared as returning non-void either contains a `return` statement that is missing an expression or control may reach the end of the function without a value being returned.
Supports MISRA C 2004 Rule 16.8 (Req)
Supports MISRA C 2012 Rule 17.4 (Mand)
Supports MISRA C++ Rule 8-4-3 (Req)

534 **ignoring return value of function *symbol***
warning
A function that returns a value is called just for side effects as, for example, in a statement by itself or the left-hand side of a comma operator. Try: `(void) function();` to call a function and ignore its return value.
Supports MISRA C 2004 Rule 16.10 (Req)
Supports MISRA C 2012 Directive 4.7 (Req)
Supports MISRA C 2012 Rule 17.7 (Req)
Supports MISRA C++ Rule 0-1-7 (Req)
Supports MISRA C++ Rule 0-3-2 (Req)

537 **repeated include file '*file*'**
warning
The file whose inclusion within a module is being requested has already been included in this compilation. The file is processed normally even if the message is given. If it is your standard practice to repeat included files then simply suppress this message.

539 **unexpected positive indentation**
warning
The current line was found to be positively indented from a clause that did not control the line in question. For example:

```
if( n > 0 )
    x = 3;
    y = 4;
```

will result in this warning being issued for `y = 4;`.

540 **initializer-string for char array is too long**
warning
A string initializer required more space than what was allocated.

541 ***hex/octal* escape sequence out of range**
warning
The size of a character constant specified with `\x`*ddd* or `\x`*hhh* equaled or exceeded `2**b` where `b` is the number of bits in a byte (established by the `-sb` option). The default is `-sb8`.

542 **excessive size for bit field**
warning
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 ***string* directive not followed by EOL**
warning
The preprocessor directive `#endif` should be followed by an end-of-line. Some compilers specifically allow commentary to follow the `#endif`. If you are following that convention simply turn this error message off.
Supports MISRA C 2004 Rule 19.16 (Req)
Supports MISRA C 2012 Rule 20.3 (Req)

Supports MISRA C 2012 Rule 20.13 (Req)
Supports MISRA C++ Rule 16-0-8 (Req)

545 taking address of array
warning
An attempt was made to take the address of an array name. At one time such an expression was officially illegal (K&R C [5]), was not consistently implemented, and was, therefore, suspect. However, the expression is legal in ANSI/ ISO C and designates a pointer to an array. For example, given

```
int a[10];
int (*p) [10];
```

Then `a` and `&a`, as pointers, both represent the same bit pattern, but whereas `a` is a pointer to `int`, `&a` is a pointer to an array of 10 integers. Of the two only `&a` may be assigned to `p` without complaint. If you are using the `&` operator in this way, we recommend that you disable this message.

546 explicitly taking address of function
warning
An attempt was made to take the address of a function name. Since names of functions by themselves are promoted to address, the use of the `&` is redundant and could be erroneous.

547 redefinition of macro *name* conflicts with previous definition
warning
The indicated symbol had previously been defined (via `#define`) to some other value.
Supports MISRA C 2012 Rule 5.4 (Req)

548 if statement has no body or else
warning
A construct of the form `if(`*e*`);` was found, and it was not followed by an `else`. This is almost certainly an unwanted semi-colon as it inhibits the `if` from having any effect.

549 explicit cast from *type* to *type*
warning
A cast was made from a pointer to some enumerated type or from an enumerated type to a pointer. This is probably an error. Check your code and if this is not an error, then cast the item to an intermediate form (such as an `int` or a `long`) before making the final cast.

550 local variable *symbol* declared in *symbol* not subsequently accessed
warning
A variable (local to some function) was not accessed. This means that the value of a variable was never used. Perhaps the variable was assigned a value but was never used. Note that a variable's value is not considered accessed by autoincrementing or autodecrementing unless the autoincrement/decrement appears within a larger expression, which uses the resulting value. The same applies to a construct of the form: *var* `+=` *expression*. If an address of a variable is taken, its value is assumed to be accessed. However, casting that address to a non-pointer causes Lint to forget this sense of "accessed-ness." An array, `struct` or `union` is considered accessed if any portion thereof is accessed.
Supports MISRA C++ Rule 0-1-4 (Req)

551 static variable *symbol* not accessed
warning
A variable (declared `static` at the module level) was not accessed though the variable was referenced. See the explanation under message `550` (above) for a description of "access".
Supports MISRA C++ Rule 0-1-4 (Req)

552 **external variable *symbol* not accessed**
warning An external variable was not accessed though the variable was referenced. See the explanation under message 550 above for a description of "access".
Supports MISRA C++ Rule 0-1-4 (Req)

553 **undefined preprocessor variable *name*, assumed 0**
warning The indicated variable had not previously been defined within a `#define` statement and yet it is being used in a preprocessor condition of the form `#if` or `#elif`. Conventionally all variables in preprocessor expressions should be pre-defined. The value of the variable is assumed to be `0`.
Supports MISRA C 2004 Rule 19.11 (Req)
Supports MISRA C 2012 Rule 20.9 (Req)
Supports MISRA C++ Rule 16-0-7 (Req)

555 **#elif not K&R**
warning The `#elif` directive was used and the K&R preprocessor flag (`+fkp`) was set. Either do not set the flag or do not use `#elif`.

557 **invalid conversion specifier '*string*'**
warning The format string supplied to a `printf`/`scanf` style function was not recognized. It is neither a standard format nor a recognized common extension (see message 816).

558 **too few data arguments for format string (*integer* missing)**
warning The number of arguments supplied to a `printf`/`scanf` style function was less than the number expected. The number of missing arguments is given by *integer*. See also message 719.

559 **format '*string*' specifies type *type* which is inconsistent with argument no. *integer* of *string type***
warning The argument corresponding to a conversion specifier in a `printf`/`scanf` style function was not of the correct type. The format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
extern char * buffer;
sprintf(buffer, "%f", 371);
```

will result in the message:

```
format '\%f' specifies type 'double' which is inconsistent with
argument no. 3 of type 'int'
```

For conflicting integer types that differ only in signedness (e.g. `int` vs. `unsigned int`) or exact type (e.g. `int` vs. `long` when both are the same size) 705 is given instead. For conflicting pointer to integer types that differ only in the signedness or exact type of the pointee, 706 is given.

563 **label *name* not referenced**
warning *name* appeared as a label but there was no statement that referenced this label.
Supports MISRA C 2012 Rule 2.6 (Adv)

564 **variable *symbol* depends on order of evaluation**
warning The named variable was both modified and accessed in the same expression in such a way that the result

depends on whether the order of evaluation is left-to-right or right-to-left. One such example is: `n + n++` since there is no guarantee that the first access to `n` occurs before the increment of `n`. This message is also triggered by the potential modification of an object by a call to a function whose corresponding parameter is a non-`const` reference or a non-`const` pointer. Volatile variables are also checked for repeated use in an expression.
Supports MISRA C 2004 Rule 12.2 (Req)
Supports MISRA C 2012 Rule 13.2 (Req)
Supports MISRA C++ Rule 5-0-1 (Req)

565 **declaration of tag *type* will not be visible outside of this function**
warning The named tag appeared in a prototype or in an inner block and was not previously seen in an outer (file-level) scope. Declare the tag before the function if the intention is for it to be visible outside the function.

566 **inconsistent or redundant *length modifier/flag* '*string*' used with '*string*' conversion specifier**
warning This message is given for format specifiers within formats for the `printf`/`scanf` family of functions. The indicated *length modifier* or *flag character* found in a format specifier either has no effect or is not allowed to be combined with the provided conversion specifier. For example:

```
printf("%+u", 123);
```

will yield the message:

```
inconsistent or redundant flag '+' used with 'u' conversion specifier
```

because the `+` flag is valid only with signed conversions, its use with other conversions results in undefined behavior.

567 **expected minimum field width for flag '*string*'**
warning This message is given for format specifiers within formats for the `printf`/`scanf` family of functions. A numeric field or asterisk was expected at a particular point in the scanning of the format. For example: `%-d` requests left justification of a decimal integer within a format field. But since no field width is given, the request is meaningless.

568 **nonnegative quantity is never less than zero**
warning Comparisons of the form:

```
u >= 0       0 <= u
u <  0       0 >  u
```

are suspicious if `u` is an unsigned quantity or a quantity judged to be never less then 0. See also message 775.

569 **loss of information (*context*) in implicit conversion from *type integer* (*integer* bits) to *type* (*integer* bits)**
warning An assignment (or implied assignment, see *context*) was made from a constant to an integral variable that is not large enough to hold the constant. Examples include placing a hex constant whose bit requirement is such as to require an `unsigned int` into a variable typed as `int`. The number of bits given does not count the sign bit.

570 **negative *type integer* loses sign during implicit conversion (*context*) to *type***
warning An assignment (or implied assignment, see *context*) is being made from a negative constant into an unsigned

quantity. Casting the constant to `unsigned` will remove the diagnostic but is this what you want? If you are assigning all ones to an `unsigned`, remember that `~0` represents all ones and is more portable than `-1`.

571 warning **cast from *type* to *type* results in sign extension**

Usually this warning is issued for casts of the form:

```
(unsigned) ch
```

where `ch` is declared as `char` and `char` is signed. Although the cast may appear to prevent sign extension of `ch`, it does not. Following the normal conversion rules of C, if `ch` is negative then it cannot be representd in an `unsigned` type and so a quantity of `2**`*n* is added to the signed quantity where *n* is the number of bits in the destination. If `2**`*m* were added, where *m* is the number of bits in the source, i.e. `ch`, then the sign extension would not occur. To suppress sign extension you may use:

```
(unsigned char) ch
```

Otherwise, if sign extension is what you want and you just want to suppress the warning in this instance you may use:

```
(unsigned) (int) ch
```

Although these examples have been given in terms of casting a `char`, this message will also be given whenever this cast is made upon a signed quantity whose size is less than the casted type. Examples include signed bit fields, expressions involving `char`, and expressions involving `short` when this type is smaller than `int` or a direct cast of an `int` to an `unsigned long` (if `int` is smaller than long). This message is not issued for constants or for expressions involving bit operations.

572 warning **excessive shift value (precision *integer* shifted right by *integer*)**

A quantity is being shifted to the right whose precision is equal to or smaller than the shifted value. For example,

```
ch >> 10
```

will elicit this message if `ch` is typed `char` and where `char` is less than 10 bits wide (the usual case). To suppress the message in this case you may cast the shifted quantity to a type whose length is at least the length of the shift value.

The precision of a constant (including enumeration constants) is determined from the number of bits required in its binary representation. The precision does not change with a cast so that `(unsigned) 1 >> 3` still yields the message. But normally the only way an expression such as `1 >> 3` can legitimately occur is via a macro. In this case use `-emacro`.

573 warning **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 warning **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 warning **enumeration constant exceeds range for integers**

For many compilers the value of an enumeration constant is limited to those values that can fit within a signed or unsigned int.

576 warning **excess elements in *string* initializer**

In a brace-enclosed initializer, there are more items than there are elements of the aggregate, which will result in undefined behavior as this is a constraint violation in C. For example:

```
int array[3] = {1, 2, 3, 4};
```

In C++, an error is emitted instead.
Supports MISRA C 2004 Rule 9.2 (Req)

578 warning **declaration of *symbol* hides *string***

A local symbol has the identical name as a variable or field specified by *detail.* This could be dangerous. Was this deliberate? It is usually best to rename the local symbol.
Supports MISRA C 2004 Rule 5.2 (Req)
Supports MISRA C 2012 Rule 5.3 (Req)
Supports MISRA C++ Rule 2-10-2 (Req)

579 **parameter preceding ellipsis cannot be '*string*'**
warning
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. Attempting to extract the variable arguments from a call to such a function results in undefined behavior.

580 **redeclaration of function *symbol* causes loss of prototype**
warning
A declaration of a function within a block hides a declaration in an outer scope in such a way that the inner declaration has no prototype and the outer declaration does. A common misconception is that the resulting declaration is a composite of both declarations but this is only the case when the declarations are in the same scope not within nested scopes. If you do not care about prototypes you may suppress this message. You will still receive other type-difference warnings.

583 **comparing type '*type*' with EOF**
warning
This message is issued when some form of character is compared against the `EOF` macro. `EOF` is normally defined to be `-1`. For example:

```
while( (ch = getchar()) != EOF ) ...
```

If `ch` is defined to be an `int` all is well. If however it is defined to be some form of `char`, then trouble might ensue. If `ch` is an `unsigned char` then it can never equal `EOF`. If `ch` is a `signed char` then you could get a premature termination because some data character happened to be all ones.

Note that `getchar` returns an `int`. The reason it returns an `int` and not a `char` is because it must be capable of returning 257 different values (256 different characters plus `EOF`, assuming an 8-bit character). Once this value is assigned to a `char` only 256 values are then possible – a clear loss of information.

584 **trigraph sequence (??*character*) detected**
warning
This message is issued whenever a trigraph sequence is detected and the trigraph processing has been turned off (with a `-ftg`). If this is within a string (or character) constant then the trigraph nature of the sequence is ignored. That is, three characters are produced rather than just one. This is useful if your compiler does not process trigraph sequences and you want linting to mirror compilation. Outside of a string we issue this warning but we do translate the sequence since it cannot make syntactic sense in its raw state.
Supports MISRA C 2004 Rule 4.2 (Req)
Supports MISRA C 2012 Rule 4.2 (Adv)
Supports MISRA C++ Rule 2-3-1 (Req)

585 **the sequence (??*character*) is not a valid trigraph sequence**
warning
This warning is issued whenever a pair of '`?`' characters is seen within a string (or character) constant but that pair is not followed by a character, which would make the triple a valid Trigraph sequence. Did the programmer intend this to be a Trigraph sequence and merely err? Even if no Trigraph were intended it can easily be mistaken by the reader of the code to be a Trigraph. Moreover, what assurances do we have that in the future the invalid Trigraph might not become a valid Trigraph and change the meaning of the string? To protect yourself from such an event you may place a backslash between the '`?`' characters. Alternatively you may use concatenation of string constants. For example:

```
pattern = "(???) ???-????";           // warning 585
pattern = "(?\?\?) ?\?\?-?\?\?\?";    // no warning
#define Q "?"
pattern="(" Q Q Q ") " Q Q Q "-" Q Q Q Q;        //no warning
```

586 ***string* '*name*' is deprecated. *string***
warning
The *name* has been deprecated by some use of the deprecate option. See `-deprecate`. The first *string* is one of the allowed categories of deprecation. The trailing *string* is part of the deprecate option and should explain why the facility has been deprecated.
Supports MISRA C 2004 Rule 2.1 (Req)
Supports MISRA C 2004 Rule 20.4 (Req)
Supports MISRA C 2004 Rule 20.5 (Req)
Supports MISRA C 2004 Rule 20.6 (Req)
Supports MISRA C 2004 Rule 20.7 (Req)
Supports MISRA C 2004 Rule 20.8 (Req)
Supports MISRA C 2004 Rule 20.10 (Req)
Supports MISRA C 2004 Rule 20.11 (Req)
Supports MISRA C 2004 Rule 20.12 (Req)
Supports MISRA C 2012 Directive 4.12 (Req)
Supports MISRA C 2012 Rule 8.14 (Req)
Supports MISRA C 2012 Rule 21.3 (Req)
Supports MISRA C 2012 Rule 21.4 (Req)
Supports MISRA C 2012 Rule 21.5 (Req)
Supports MISRA C 2012 Rule 21.6 (Req)
Supports MISRA C 2012 Rule 21.7 (Req)
Supports MISRA C 2012 Rule 21.8 (Req)
Supports MISRA C 2012 Rule 21.9 (Req)
Supports MISRA C 2012 Rule 21.10 (Req)
Supports MISRA C 2012 Rule 21.12 (Req)
Supports MISRA C++ Rule 17-0-5 (Req)
Supports MISRA C++ Rule 18-0-2 (Req)
Supports MISRA C++ Rule 18-0-3 (Req)
Supports MISRA C++ Rule 18-0-5 (Req)
Supports MISRA C++ Rule 18-2-1 (Req)
Supports MISRA C++ Rule 8-4-1 (Req)
Supports MISRA C++ Rule 19-3-1 (Req)

587 **predicate '*string*' can be pre-determined and always evaluates to *true/false***
warning
The predicate, identified by *string*, cannot possibly be other than what is indicated by the message. For example:

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

would be greeted with the message that '`==`' always evaluates to '`false`'.

See Precision, Viable Bit Patterns, and Representable Values for more information.
Supports MISRA C++ Rule 0-1-9 (Req)

592 **non-literal format specifier used without arguments**
warning
A `printf`/`scanf` style function received a non-literal format specifier without trailing arguments. For example:

```
char msg[100];
...
printf( msg );
```

This can easily be rewritten to the relatively safe:

```
char msg[100];
```

```
...
printf( "%s", msg );
```

The danger lies in the fact that `msg` can contain hidden format codes. If `msg` is read from user input, then in the first example, a naive user could cause a glitch or a crash and a malicious user might exploit this to undermine system security. Since the unsafe form can easily be transformed into the safe form, the latter should always be used.

593 **custodial pointer *symbol* possibly not freed nor returned**
warning
This is the 'possible' version of message 429. A pointer of `auto` storage class was allocated storage and not all paths leading to a `return` statement or to the end of the function contained either a `free` or a `return` of the pointer. Hence there is a potential memory leak. For example:

```
void f(int n) {
    int *p = new int;
    if (n) delete p;
}                       //message 593
```

In this example an allocation is made and, if `n` is 0, no `delete` will have been made.

Please see message 429 for an explanation of "custodial" and ways of regulating when pointer variables retain custody of allocations.

597 **suspicious use of unary operator could be confused for compound assignment (*string*)**
warning
A construct such as:

```
a =- b;
```

or

```
a =+ b;
```

which is suspect: did the programmer intend to use the `-=/+=` compound assignment operator? The message is only issued when there is no space between the `=` and the `-/+` and when there is a space between the `-/+` and its operand. '*string*' contains the form of compound assignment that the expression may be confused for.

598 **excessive shift value (precision '*integer*' shifted by '*integer*')**
warning
A quantity is being shifted to the left by a value greater than or equal to the precision of that quantity or by a negative value. For example,

```
i << 32
```

will elicit this message if `i` is typed `int` and where `int` is 32 bits wide or less (the usual case). Such shift results in undefined behavior. To suppress the message you may cast the shifted quantity to a type whose length is at least the length of the shift value.
Supports MISRA C 2012 Rule 12.2 (Req)

599 **cannot open file matching wild card pattern '*string*'**
warning
A wild card pattern was used where the name of a file was expected but there were no files found that match the given pattern so it will be ignored. '*string*' contains the offending pattern.

601 **expected a type, int assumed**
warning
A declaration did not have an explicit type. `int` was assumed. Was this a mistake? This could easily happen if an intended comma was replaced by a semicolon. For example, if instead of typing:

```
double      radius,
            diameter;
```

the programmer had typed:

```
double      radius;
            diameter;
```

this message would be raised.
Supports MISRA C 2004 Rule 8.2 (Req)
Supports MISRA C 2012 Rule 8.1 (Req)

602 **'/*' within block comment**
warning
The sequence `/*` was found within a comment. Was this deliberate? Or was a comment end inadvertently omitted? If you want PC-lint Plus to recognize nested comments you should set the Nested Comment flag using the `+fnc` option. Then this warning will not be issued. If it is your practice to use the sequence:

```
/*
/*              */
```

then use `-e602`.
Supports MISRA C 2004 Rule 2.3 (Req)
Supports MISRA C 2004 Rule 2.4 (Adv)
Supports MISRA C 2012 Directive 4.4 (Adv)
Supports MISRA C 2012 Rule 3.1 (Req)
Supports MISRA C++ Rule 2-7-1 (Req)

603 **argument to parameter of type pointer to const may be a pointer to uninitialized memory**
warning
The address of the named symbol is being passed to a function where the corresponding parameter is declared as pointer to `const`. This implies that the function will not modify the object. If this is the case then the original object should have been initialized sometime earlier.

604 **returning address of auto variable *symbol***
warning
The address of the named symbol is being passed back by a function. Since the object is an `auto` and since the duration of an `auto` is not guaranteed past the `return`, this is most likely an error. You may want to copy the value into a global variable and pass back the address of the global or you might consider having the caller pass an address of one of its own variables to the callee.
Supports MISRA C 2004 Rule 17.6 (Req)
Supports MISRA C 2012 Rule 18.6 (Req)
Supports MISRA C++ Rule 7-5-1 (Req)

605 **pointee implicitly *gains/loses const/volatile* qualifier in conversion from *type* to *type* (*context*)**
warning
This warning is typically caused by assigning a (pointer to `const`) to an ordinary pointer. For example:

```
int *p;
const int *q;
p = q;     /* 605   */
```

The message will be inhibited if a cast is used as in:

```
p = (int *) q;
```

An increase in capability is indicated because the `const` pointed to by `q` can now be modified through `p`. This message can be given for the `volatile` qualifier as well as the `const` qualifier and may be given for arbitrary pointer depths (pointers to pointers, pointers to arrays, etc.).

If the number of pointer levels exceeds one, things get murky in a hurry. For example:

```
const char ** ppc;
char ** pp;
pp = ppc;        /* 605 - clearly not safe */
ppc = pp;        /* 605 - looks safe but it's not */
```

The problem is that after the above assignment, a pointer to a `const char` can be assigned indirectly through `ppc` and accessed through `pp`, which can then modify the `const char`.

The message speaks of an "increase in capability" in assigning to `ppc`, which seems counter intuitive because the indirect pointer has less capability. However, assigning the pointer does not destroy the old one and the combination of the two pointers represents a net increase in capability.

The message may also be given for function pointer assignments when the prototype of one function contains a pointer of higher capability than a corresponding pointer in another prototype. There is a curious inversion here whereby a prototype of lower capability translates into a function of greater trust and hence greater capability (a Trojan Horse). For example, let

```
void warrior( char * );
```

be a function that destroys its argument. Consider the function:

```
void Troy( void (*horse)(const char *) );\\
```

`Troy()` will call `horse()` with an argument that it considers precious (i.e. not to be modified) believing the `horse()` will do no harm. Before compilers knew better and believing that adding in a `const` to the destination never hurt anything, earlier compilers allowed the Greeks to pass `warrior()` to `Troy` and the rest, as they say, is history.
Supports MISRA C++ Rule 9-3-1 (Req)

606 **non-ANSI escape sequence: '*string*'**
warning An escape sequence occurred, within a character or string literal, that was not on the approved list, which is:

```
\' \" \?  \\\a \b \f \n \r \t \v
```
octal-digits x*hex-digits*

Supports MISRA C 2004 Rule 4.1 (Req)
Supports MISRA C++ Rule 2-13-1 (Req)

608 **assigning to array parameter *symbol***
warning An assignment is being made to a parameter that is typed array. For the purpose of the assignment, the parameter is regarded as a pointer. Normally such parameters are typed as pointers rather than arrays. However if this is your coding style you should suppress this message.

611 **cast between pointer to function type *type* and pointer to object type *type***
warning Either a pointer to a function is being cast to a pointer to an object or vice versa. This is regarded as questionable by the language standards. If this is not a user error, suppress this warning.
Supports MISRA C++ Rule 5-2-6 (Req)

612 **declaration does not declare anything**
warning A declaration contained just a storage class and a type. This is almost certainly an error since the only

time a type without a declarator makes sense is in the case of a `struct`, `union` or `enum` but in that case you wouldn't use a storage class.

613 potential use of null pointer *symbol*
warning
From information gleaned from earlier statements, it is possible that a null pointer (a pointer whose value is 0) can be used in a context where null pointers are inappropriate. Information leading to this determination is provided as a series of supplemental messages. See also message 413.

614 auto aggregate initializer not constant
warning
Prior to C99, auto aggregate initialization could consist only of constant expressions. This message is only issued in C89/C90 mode. See also message 446.

615 auto aggregate initializer has side effects
warning
This warning is similar to `614`. Auto aggregates (arrays, structures and possibly unions) are normally initialized by a collection of constant expressions without side-effects. If your compiler supports side-effects in the initializers of aggregate, you may want to suppress this message. This message is only issued in C89/C90 mode.

616 control flow falls through to next case without an intervening comment
warning
It is possible for flow of control to fall into a case statement or a `default` statement from above. Was this deliberate or did the programmer forget to insert a `break` statement? If this was deliberate then place a comment immediately before the statement that was flagged as in:

```
case'a': a=0;
/* fall through */
            case 'b':  a++;
```

Note that the message will not be given for a `case` that merely follows another `case` without an intervening statement. Also, there must actually be a possibility for flow to occur from above. See also message 825 and option `-fallthrough`.

618 storage class specified after a type
warning
A storage class specifier (`static`, `extern`, `typedef`, `register` or `auto`) was found after a type was specified. This is legal but unusual. Either place the storage class specifier before the type or suppress this message.

620 suspicious constant (L or one?)
warning
A constant ended in a lower-case letter '`l`'. Was this intended to be a one? The two characters look very similar. To avoid misinterpretations, use the upper-case letter '`L`'.
Supports MISRA C 2012 Rule 7.3 (Req)

621 identifier clash (*string*): *string* '*string*' clashes with *string* '*string*'
warning
The `-idlen` option can be used to specify the number of *significant characters* in *external*, *preprocessor*, and *preprocessor* names. Names that are not unique within the initial characters specified by this option are said to "clash" and are reported by this message.

For the purpose of this message, identifiers are classified as *external* (function and variable names with external linkage), *preprocessor* (macro names and macro parameter names), and *compiler* (all other identifiers, including those with internal and no linkage, such as fields, tags, enumeration constants, typedefs, labels, etc). The type of clash is reported by the first *string* parameter. The possible values of this parameter and the cases in which the clashes are reported are detailed in the following list.

- `field vs field`
  The names of two fields clash within the same structure or union.
- `tag vs tag`
  The names of two struction, union, or enum tags clash within a single translation unit.
- `label vs label`
  The names of two labels clash within a single function.
- `internal vs internal, same scope`
  Two *compiler* identifiers clash in the same scope.
- `internal vs external, same scope`
  A *compiler* identifier clashes with an *external* identifier in the same scope.
- `external vs internal, same scope`
  An *external* identifier clashes with a *compiler* identifier in the same scope.
- `internal vs internal, enclosing scope`
  A *compiler* identifier clashes with another *compiler* identifier in an enclosing scope.
- `internal vs external, enclosing scope`
  A *compiler* identifier clashes with an *external* identifier in an enclosing scope.
- `external vs internal, enclosing scope`
  An *external* identifier clashes with a *compiler* identifier in an enclosing scope.
- `external vs external`
  Two *external* identifiers clash (anywhere in the analyzed program).
- `macro vs macro`
  The names of two macros in the same translation unit clash.
- `macro vs macro parameter`
  The name of a macro clashes with the name of a macro parameter of a currently defined macro.
- `macro parameter vs macro parameter`
  The name of a macro parameter clashes with the name of a parameter of the same macro.

Fields, tags, and labels each exist in their own name spaces and thus never clash with identifiers in other name spaces. *Internal* here refers to *compiler* identifiers that are not field, tag, or label identifiers. For clashes between internal and external names, the number of significant characters for *compiler* identifiers is used to determine a clash. Clashes between two *external* identifiers are reported regardless of scope. External identifiers in separate modules that clash are reported during global wrapup.

This message is not issued for C++ modules (all characters in identifier names are significant and case-sensitive in C++) or for identifiers with identical spelling.
Supports MISRA C 2012 Rule 5.1 (Req)
Supports MISRA C 2012 Rule 5.2 (Req)
Supports MISRA C 2012 Rule 5.3 (Req)
Supports MISRA C 2012 Rule 5.4 (Req)

629 **'static' function declaration at block scope is non standard**
warning A `static` storage class specifier was found for a function declaration within a function. The `static` storage class specifier is permitted only for functions in declarations that have file scope (i.e., outside any function). Either move the declaration outside the function or change `static` to `extern`; if the second choice is made, make sure that a `static` declaration at file scope also exists before the `extern` declaration.

631 **tag *symbol* defined differently at *location***
warning The textttstruct, `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 **strong type mismatch: assigning '*strong-type*' to '*strong-type*' in context '*context*'**
warning An assignment (or implied assignment, *context* indicates which), violates a Strong type check as requested by a `-strong(A...)` option. See Chapter 7 Strong Types.

633 **strong type mismatch: extracting '*strong-type*' into '*strong-type*' in context '*context*'**
warning An assignment (or implied assignment, *context* indicates which), violates a Strong type check as requested by a `-strong(X...)` option. See Chapter 7 Strong Types.

634 **strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**
warning An equality operation (`==` or `!=`) or a conditional operation (`? :`) violates a Strong type check as requested by a `-strong(J...)` option. This message would have been suppressed using flags `"Je"`. See Chapter 7 Strong Types.

635 **changing the *parent/index* type of '*strong-type*' from '*strong-type*' to '*strong-type*'**
warning The strong parent of the *symbol* is being reset. This is being done with a `-parent` option or by a `typedef`. Note that this may not necessarily be an error; you are being alerted to the fact that the old link is being erased. See Chapter 7 Strong Types.

636 **strong type difference: pointees are '*strong-type*' and '*strong-type*'**
warning Pointers are being compared and there is a strong type clash below the first level. For example,

```
/*lint -strong(J, INT) */
typedef int INT;
INT *p;  int *q;

if( p == q ) /*  Warning 636  */
```

will elicit this warning. This message would have been suppressed using flags `"Je"` or `"Jr"` or both. See Chapter 7 Strong Types.

637 **strong type mismatch: '*strong-type*' is not an acceptable index type for '*strong-type*' (expected '*strong-type*')**
warning This is the message you receive when an inconsistency with the `-index` option is recognized. A subscript is not the stipulated type (the first type mentioned in the message) nor equivalent to it within the hierarchy of types. See Chapter 7 Strong Types and also `+fhx`.

638 **strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**
warning A relational operation (`>= <= > <`) violates a Strong type check as requested by a `-strong(J...)` option. This message would have been suppressed using flags `"Jr"`. See Chapter 7 Strong Types.

639 **strong type mismatch: cannot join '*strong-type*' and '*strong-type*' using operator '*operator*'**
warning A binary operation other than an equality or a relational operation violates a Strong type check as requested

by a `-strong(J...)` option. This message would have been suppressed using flags "`Jo`". See Chapter 7 Strong Types.

640 **warning** **strong type mismatch: '*strong-type*' is not an acceptable boolean type for *parameter* '*keyword*' statement (expected '*strong-type*')**
A Boolean context expected a type specified by a `-strong(B...)` option. See Chapter 7 Strong Types.

641 **warning** **implicit conversion of enum *symbol* to integral type *type***
An enumeration type was used in a context that required a computation such as an argument to an arithmetic operator or was compared with an integral argument. This warning will be suppressed if you use the integer model of enumeration (`+fie`) but you will lose some valuable type-checking in doing so. An intermediate policy is to simply turn off this warning. Assignment of `int` to `enum` will still be caught.

This warning is not issued for a tagless `enum` without variables. For example

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

This cannot be used as a separate type. PC-lint Plus recognizes this and treats `false` and `true` as arithmetic constants.

644 **warning** **potentially using an uninitialized value**
An `auto` variable was not necessarily assigned a value before use.
Supports MISRA C 2004 Rule 9.1 (Req)
Supports MISRA C 2012 Rule 9.1 (Mand)

645 **warning** **potentially passing an uninitialized value to a const parameter**
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.

646 **warning** **'*string*' within '*string*' loop; may have been misplaced**
A `case` or `default` statement was found within a `for`, `do`, or `while` loop. Was this intentional? At the very least, this reflects poor programming style.
Supports MISRA C++ Rule 15-0-3 (Req)

647 **warning** **possible truncation before conversion from *type* to *type***
This message is issued when it appears that there may have been an unintended loss of information during an operation involving `int` or `unsigned int` the result of which is later converted to `long`. It is issued only for systems in which `int` is smaller than `long`. For example:

```
(long) (n << 8)
```

might elicit this message if `n` is `unsigned int`, whereas

```
(long) n << 8
```

would not. In the first case, the shift is done at `int` precision and the high order 8 bits are lost even though there is a subsequent conversion to a type that might hold all the bits. In the second case, the shifted bits are retained.

The operations that are scrutinized and reported upon by this message are: shift left, multiplication, and bit-wise complementation. Addition and subtraction are covered by Informational message 776.

The conversion to `long` may be done explicitly with a cast as shown or implicitly via assignment, return, argument passing or initialization.

The message can be suppressed by casting. You may cast one of the operands so that the operation is done in full precision as is given by the second example above. Alternatively, if you decide there is really no problem here (for now or in the future), you may cast the result of the operation to some form of `int`. For example, you might write:

```
(long) (unsigned) (n << 8)
```

In this way PC-lint Plus will know you are aware of and approve of the truncation.

648 warning **overflow in computing constant for operation '*operator*'**

Arithmetic overflow was detected while computing a constant expression. For example, if `int` is 16 bits then `200 * 200` will result in an overflow. *operator* 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) 0xFFFu << 8     /* 648 */
```

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

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

Supports MISRA C 2004 Rule 12.11 (Adv)
Supports MISRA C 2012 Rule 12.4 (Adv)
Supports MISRA C++ Rule 5-19-1 (Adv)

649 warning **right shifting a negative constant expression has implementation defined behavior**

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 warning **constant '*integer*' 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 was used in a way that is not appropriate for the constraints on the value of the other operand. For example, if 300 is compared against a `char` variable, this warning will be issued. Moreover, if `char` is signed (and 8 bits) you will get this message if you compare against an integer greater than 127. The problem can be fixed with a cast. For example:

```
if( ch == 0xFF ) ...
if( (unsigned char) ch == 0xFF ) ...
```

If `char` is signed (`+fcu` has not been set) the first receives a warning and can never succeed. The second suppresses the warning and corrects the bug.

PC-lint Plus will take into account the limited precision of some operands such as bit-fields and enumerated types. Also, PC-lint Plus will take advantage of computations that limit the precision of an operand. For example,

```
if( (n & 0xFF) >> 4 == 16 ) ...}
```

will receive this warning because the left-hand side is limited to 4 bits of precision.

See Precision, Viable Bit Patterns, and Representable Values for more information. See also message `2650` for constants that are out of range for only part of a compound comparison operator.
Supports MISRA C 2004 Rule 13.7 (Req)
Supports MISRA C 2012 Rule 14.3 (Req)

651 inconsistent bracing in aggregate initialization
warning An initializer for a complex aggregate is being processed that contains some subaggregates that are bracketed and some that are not. ANSI/ISO recommends either "minimally bracketed" initializers in which there are no interior braces or "fully bracketed" initializers in which all interior aggregates are bracketed.

652 #define of macro '*string*' with same name as previously declared symbol *symbol*
warning A macro is being defined for a symbol that had previously been declared. For example:

```
int n;
#define n N
```

will draw this complaint. Prior symbols checked are local and global variables, functions and `typedef` symbols, and `struct`, `union` and `enum` tags. Not checked are `struct` and `union` members.

653 result of integer division being converted to *type*
warning When two integers are divided and assigned to a floating point variable the fraction portion is lost. For example, although

```
double x = 5 / 2;
```

appears to assign `2.5` to `x` it actually assigns `2.0`. To make sure you do not lose the fraction, cast at least one of the operands to a floating point type. If you really wish to do the truncation, cast the resulting divide to an integral (`int` or `long`) before assigning to the floating point variable.

654 option '*option*' is obsolete; *detail*
warning The specified *option* is obsolete and should no longer be used. The *detail* parameter contains additional information such as further explanation or alternatives.

655 bitwise operation uses compatible enums (of type enum *type*)
warning A bit-wise operator (one of '`|`', '`&`' or '`^`') is used to combine two compatible enumerations. The type of the result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

656 arithmetic operation uses compatible enums (of type enum *type*)
warning An arithmetic operator (one of '`+`', or '`-`') is used to combine two compatible enumerations. The type of the

result is considered to be the enumeration. This is considered a very minor deviation from the strict model and you may elect to suppress this warning.

657 unusual (nonportable) anonymous struct or union
warning
A `struct` or `union` declaration without a declarator was taken to be anonymous. However, the anonymous `union` supported by C++ and other dialects of C require untagged unions. Tagged unions and tagged or untagged structs are rarely supported, as anonymous.

658 anonymous union assumed (use flag +fan)
warning
A union without a declarator was found. Was this an attempt to define an anonymous `union`? If so, anonymous unions should be activated with the `+fan` flag. This flag is activated automatically for C++.

660 option '*string*' requests removing an extent that is not on the list
warning
A number of options use the '-' prefix to remove and the '+' prefix to add elements to a list. For example to add (the most unusual) extension `.C++` to designate C++ processing of files bearing that extension, a programmer should employ the option:

```
+cpp(.C++)
```

However, if a leading '-' is employed (a natural mistake) this warning will be emitted.

661 potential out of bounds pointer access: excess of *integer* byte(s)
warning
An out-of-bounds pointer may have been accessed. See message 415 for a description of the *integer* parameter. For example:

```
int a[10];
if( n <= 10 ) a[n] = 0;
```

Here the programmer presumably should have written `n < 10`. This message is similar to message 415 but differs from it by the degree of probability. See Chapter 8 Value Tracking.
Supports MISRA C 2012 Rule 18.1 (Req)
Supports MISRA C++ Rule 5-0-16 (Req)

662 possibly creating out-of-bounds pointer: excess of *integer* byte(s)
warning
An out-of-bounds pointer may have been created. See message 415 for a description of the *integer* parameter. For example:

```
int a[10];
if( n <= 20 ) f( a + n );
```

Here, it appears as though an illicit pointer is being created, but PC-lint Plus cannot be certain. See also message 416 and Chapter 8 Value Tracking.
Supports MISRA C 2012 Rule 18.1 (Req)
Supports MISRA C++ Rule 5-0-16 (Req)

663 array-to-pointer decay causes indirection through first element
warning
This warning occurs in the following kind of situation:

```
struct x { int a; } y[2];
...  y->a  ...
```

Here, the programmer forgot to index the array but the error normally goes undetected because the array reference is automatically and implicitly converted to a pointer to the first element of the array. If you really mean to access the first element use `y[0].a`

664 **left hand side of logical operator contains call to function that does not return**
warning An exiting function was found on the left hand side of an operator implying that the right hand side would never be executed. For example:

```
if( (exit(0), n == 0) || n > 2 ) ...
```

Since the exit function does not return, control can never flow to the right hand operator.

665 **unparenthesized parameter *integer* in macro '*string*' is passed an expression**
warning An expression was passed to a macro parameter that was not parenthesized. For example:

```
#define mult(a,b) (a*b)
...   mult( 100, 4 + 10 )
```

Here the programmer is beguiled into thinking that the `4+10` is taken as a quantity to be multiplied by `100` but instead results in: `100*4+10`, which is quite different. The recommended remedy ([6, Section 19.4]) is to parenthesize such parameters as in:

```
#define mult(a,b) ((a)*(b))
```

The message is not arbitrarily given for any unparenthesized parameter but only when the actual macro argument sufficiently resembles an expression and the expression involves binary operators. The priority of the operator is not considered except that it must have lower priority than the unary operators. The message is not issued at the point of macro definition because it may not be appropriate to parenthesize the parameter. For example, the following macro expects that an operator will be passed as argument. It would be an error to enclose `op` in parentheses.

```
#define check(x,op,y) if( ((x) op (y)) == 0 ) print( ... )
```

Supports MISRA C 2012 Rule 20.7 (Req)

666 **expression with side effects passed to repeated parameter *integer* of macro '*string*'**
warning A repeated parameter within a macro was passed an argument with side-effects. For example:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))

... ABS( n++ )
```

Although the `ABS` macro is correctly defined to specify the absolute value of its argument, the repeated use of the parameter `x` implies a repeated evaluation of the actual argument `n++` This results in two increments to the variable `n`. [6, Section 19.6] Any expression containing a function call is also considered to have side-effects.

668 **possibly passing null pointer to function *symbol*, *context***
warning A NULL pointer is possibly being passed to a function identified by *symbol*. The argument in question is given by *context*. The function is either a library function designed not to receive a NULL pointer or a user function dubbed so via the option `-function` or `-sem`. See Sections 9.1 Function Mimicry (-function), 9.2 Semantic Specifications and 8 Value Tracking.
Supports MISRA C 2012 Directive 4.11 (Req)

669 **possible data overrun for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**
warning
This message is for data transfer functions such as `memcpy`, `strcpy`, `fgets`, etc. when the size indicated by the first cited argument (or arguments) can possibly exceed the size of the buffer area cited by the second. The message may also be issued for user functions via the `-function` or `-sem` option. See Sections 9.1 Function Mimicry (-function), 9.2 Semantic Specifications and Chapter 8 Value Tracking.
Supports MISRA C 2012 Directive 4.11 (Req)

670 **possible access beyond array for function *symbol*, *string* (size=*string*) exceeds *string* (size=*string*)**
warning
This message is issued for several library functions (such as `fwrite`, `memcmp`, etc.) wherein there is a possible attempt to access more data than exist. For example, if the length of data specified in the `fwrite` call exceeds the size of the data specified. The function is specified by *symbol* and the arguments are identified by argument number. See Sections 9.1 Function Mimicry (-function), 9.2 Semantic Specifications and Chapter 8 Value Tracking.
Supports MISRA C 2012 Directive 4.11 (Req)

671 **possibly passing to function *symbol* a negative value (*string*) *context***
warning
An integral value that may possibly be negative is being passed to a function that is expecting only positive values for a particular argument. The message contains the name of the function (*symbol*), the questionable value (*integer*) and the argument number (*context*). The function may be a standard library function designed to accept only positive values such as `malloc` or `memcpy` (third argument), or may have been identified by the user as such through the `-function` or `-sem` options. See message **422** for an example and further explanation.
Supports MISRA C 2012 Directive 4.11 (Req)

672 **assignment to custodial pointer *symbol* possibly creates memory leak**
warning
An assignment was made to a pointer variable (designated by *symbol*), which may already be holding the address of an allocated object that had not been freed. The allocation of memory, which is not freed, is considered a 'memory leak'. The memory leak is considered 'possible' because only some lines of flow will result in a leak.

673 ***string* may not be appropriate for deallocating *string***
warning
This message indicates that a deallocation (`delete`, `delete[]`, or `free`) as specified by the first *string* parameter may be inappropriate for the data being freed. The kind of data is described in the second *string* parameter. The wording 'may not' is used to indicate that only some of the lines of flow to the deallocation show data inconsistent with the allocation. See also message **424**.

674 **returning address of auto variable *symbol* through pointer *symbol***
warning
The value held by a pointer variable contains the address of an `auto` variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

675 **no prior semantics associated with '*name*' in option '*string*'**
warning
The `-function` option is used to transfer semantics from its first argument to subsequent arguments. However it was found that the first argument *name* did not have semantics.

676 **possibly indexing before the beginning of an allocation**
warning

An integer whose value was possibly negative was added to an array or to a pointer to an allocated area (allocated by `malloc`, `operator new`, etc.). This message is not given for pointers whose origin is unknown since a negative subscript is in general legal.
Supports MISRA C 2012 Rule 18.1 (Req)

677 sizeof used within preprocessor statement
warning Whereas the use of `sizeof` during preprocessing is supported by a number of compilers it is not a part of the ANSI/ISO C or C++ standard. See Section 12.6 Preprocessor sizeof.

678 member *symbol* field length (*integer*) too small for enum precision (*integer*)
warning A bit field was found to be too small to support all the values of an enumeration (that was used as the base of the bit field). For example:

```
enum color { red, green, yellow, blue };
struct abc { enum color c:2; };
```

Here, the message is not given because the four enumeration values of `color` will just fit within 2 bits. However, if one additional color is inserted, Warning `678` will be issued informing the programmer of the undesirable and dangerous condition.

679 integer operation may be truncated before being combined with a larger pointer type
warning This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals before combining with a pointer whose precision is greater than the integral expression. For example:

```
// Assuming 4-byte ints and 8-bytes pointers (-si4 -sp8)
char *f( char *p, int n, int m ) {
    return p + (n + m); // warning 679
}
```

By the rules of C/C++, the addition `n+m` is performed independently of its context and is done at integer precision. Any overflow is ignored even though the larger precision of the pointer could easily accommodate the overflow. If, on the other hand the expression were: `p+n+m`, which parses as `(p+n)+m`, no warning would be issued.

If the expression were `p+n*m` then, to suppress the warning, a cast is needed. If `long` were the same size as pointers you could use the expression:

```
return p + ((long) n * m);
```

680 suspicious truncation in arithmetic expression converted to pointer
warning An arithmetic expression was cast to pointer. Moreover, the size of the pointer is greater than the size of the expression. In computing the expression, any overflow would be lost even though the pointer type would be able to accommodate the lost information. To suppress the message, cast one of the operands to an integral type large enough to hold the pointer. Alternatively, if you are sure there is no problem you may cast the expression to an integral type before casting to pointer. See messages `647`, `776`, `790` and `679`.

681 loop is likely not entered
warning The controlling expression for a loop (either the expression within a `while` clause or the second expression within a `for` clause) evaluates initially to 0 and so it appears as though the loop is never entered.
Supports MISRA C 2004 Rule 14.1 (Req)

Supports MISRA C 2012 Rule 2.1 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)

682 **warning** **sizeof applied to parameter *symbol* of function *symbol* whose type is a sized array will yield size of *string* instead of *string***

If a parameter is typed as an array it is silently promoted to pointer. Taking the size of such an array will actually yield the size of a pointer. Consider, for example:

```
unsigned f( char a[100] ) { return sizeof(a); }
```

Here it looks as though function `f()` will return the value 100 but it will actually return the size of a pointer, which is usually 4.
Supports MISRA C 2012 AMD1 Rule 12.5 (Mand)

683 **warning** **function '*string*' #define'd, semantics may be lost**

This message is issued whenever the name of a function with some semantic association is defined as a macro. For example:

```
#define strlen mystrlen
```

will raise this message. The problem is that the semantics defined for `strlen` will then be lost. Consider this message an alert to transfer semantics from `strlen` to `mystrlen`, using `-function(strlen, mystrlen)`. The message will be issued for built-in functions (with built-in semantics) or for user-defined semantics. The message will not be issued if the function is defined to be a function with a similar name but with underscores either appended or prepended or both. For example:

```
#define strlen __strlen}
```

will not produce this message. It will produce Info 828 instead.
Supports MISRA C 2012 Rule 21.2 (Req)

685 **warning** **relational operator '*string*' always evaluates to '*string*'**

The first *string* is one of '>', '>=', '<', or '<=' and identifies the relational operator. The second *string* is one of `true` or `false`. The message is given when an expression is compared to a constant and the precision of the expression indicates that the test will always succeed or always fail. For example,

```
char ch;
...
if( ch >= -128 ) ...
```

In this example, the precision of char `ch` is 8 bits signed (assuming the `fcu` flag has been left in the OFF state) and hence it has a range of values from -128 to 127 inclusive. Hence the test is always `true`.

Note that, technically, `ch` is promoted to `int` before comparing with the constant. For the purpose of this comparison we consider only the underlying precision. As another example, if `u` is an `unsigned int` then

```
if( (u & 0xFF) > 0xFF ) ...
```

will also raise message 685 because the expression on the left hand side has an effective precision of 16 bits.
Supports MISRA C 2004 Rule 13.7 (Req)
Supports MISRA C 2012 Rule 14.3 (Req)
Supports MISRA C++ Rule 0-1-2 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)
Supports MISRA C++ Rule 0-1-9 (Req)

686 **option '*option*' is suspicious: *detail***
warning An option is considered suspicious for one of a variety of reasons. The reason is designated by *detail*. At this writing the following reasons for issuing this message are:

`unbalanced quotes` – An option was seen with a quote character that was not balanced within that same option.

`backtick preceding non-meta character is superfluous and has been dropped` – A backtick (`) was seen before a character other than a `*` or a `?`. The use of a backtick in this fashion has no effect.

`upper case characters within extension '`*`string`*`'; these will match lower case when +fff is on; try -fff` – A file extension involving uppercase letters was seen in a `+cpp` or `+lnt` option while the `+fff` flag was active or the flag became active while there were uppercase extensions registered via `+cpp` or `+lnt`. If, for example, you intend for `.c` to indicate a C module and `.C` to indicate a C++ module, turning off the `fff` flag will help avoid unnecessary complaints from PC-lint Plus.

`extraneous characters following` *`string`* – One or more characters were seen immediately following a character that is expected to signify the end of an option, such as a closing right parenthesis. While the extraneous characters are ignored, their presence may indicate a typographical error.

`the likelihood of causing meaningless output` – An option, such as `-elib(*)`, `-wlib(0)`, or `+fce` was seen; this typically hides a problem in the PC-lint Plus configuration. When using a new configuration, it's common for a user to encounter Error messages about Library header code. (This usually does not indicate a problem with library headers.) For example, a misconfiguration of PC-lint Plus preprocessor is by far the most common source of these errors. If you merely suppress basic Syntax Errors (like error 10) and/or Fatal Errors (like error 309), the underlying configuration problem still exists; as a result, PC-lint Plus will fail to parse your code correctly (because your code depends on the aforementioned library code). The output from Lint would then seem illogical and/or meaningless. Therefore, blanket suppression options like this are highly discouraged. Instead, other aspects of the Lint configuration should be modified to make Lint's behavior more similar to that of the compiler at (or, typically, before) the point of Error.

`it is too late to use -incvar as '`*`name`*`' has already been processed as incvar` – This option (`-incvar`) is used to specify the name of the environment variable that contains a list of supplementary directories to be searched for headers. This option does not have any effect after this environment variable is processed, which occurs when processing the first module. To have an effect, the option must be moved to before the first module.

`option has no effect due to zero length zone of transition` – The `-w#` or `-wlib(#)` option was seen with the same warning level of the previously provided `-w#` or `-wlib(#)` option. Because the warning level doesn't change, there is no zone of transition and therefore no effect on the message suppression set. For example, in `-w1 +e714 -w1`, the second `-w1` does not have any effect, in particular, message 714 is not suppressed because there is no zone of transition. If the goal is to suppress all messages except for errors regardless of messages that have been enabled in the meantime, it is necessary to raise the warning level and then lower it, e.g. `-w4 -w1`.

`modifying the LINT environment variable after startup has no effect; this variable should be set before program startup` – The `-setenv` option was used to set the LINT environment variable. If this variable is set when PC-lint Plus is started, its contents are processed as options before the command line options are processed. Attempting to set or change the value of this variable after program startup has no effect.

`-max_threads option must appear before first module to have any effect` – `-max_threads` is used to specify the maximum concurrent linting threads to dispatch when performing parallel analysis. This option has no effect when it appears after the first module; move the option to before the first module is referenced to obtain the desired behavior.

`the size of an incomplete type was requested in a function semantic` – The use of `@p` was used in a user-defined function return semantic but the pointee return type was not complete at the point of the call. This is suspicious because if the type is incomplete, PC-lint Plus cannot calculate its size from the number of the type's elements. Either use `@P` to specify size in bytes or make a definition of the type visible to PC-lint Plus at the point of the call.

`include path begins with unexpanded tilde prefix` – The path provided to a `-i` option began with a tilde (`~`). This was likely intended to refer to the user's home directory, but it is interpreted literally unless expanded by the shell.

`include path is the absolute path of a file rather than a directory` – The path provided to a `-i` option was a valid absolute path but refers to a file rather than a directory.

`absolute path is not accessible` – The path provided to a `-i` option unambiguously refers to an absolute path on the current platform but the target does not exist.

`include path resembles a Windows absolute path` – The path provided to a `-i` option begins with a Windows drive letter which has no special meaning on the current platform (and would simply refer to a relative directory named after the "drive letter").

`drive-relative absolute path is not accessible on the current drive` – The path provided to a `-i` option begins with a slash on Windows. This is an "absolute" path relative to the current drive letter (at the time that the path is resolved). The target does not exist on the current drive. It is possible that the directory could be found if PC-lint Plus were launched from a different drive.

687 body of '*string*' is an unparenthesized comma operator
warning
A comma operator appeared unbraced and unparenthesized in a statement following an `if`, `else`, `while` or `for` clause. For example:

```
if( n > 0 ) n = 1,
n = 2;
```

Thus the comma could be mistaken for a semi-colon and hence be the source of subtle bugs.

If the statement is enclosed in curly braces or if the expression is enclosed in parentheses, the message is not issued.

689 apparent end of C-style comment ignored
warning
The pair of characters '`*/`' was found not within a comment. As an example:

```
void f( void*/*comment*/ );
```

This is taken to be the equivalent of:

```
void f( void* );
```

That is, an implied blank is inserted between the '`*`' and the '`/`'. To avoid this message simply place an explicit blank between the two characters.

691 suspicious use of backslash
warning
The backslash character has been used in a way that may produce unexpected results. Typically this would occur within a macro such as:

```
#define A b \ // comment
```

The coder might be thinking that the macro definition will be continued on to the next line. The standard indicates, however, that the newline will not be dropped in the event of an intervening comment. This should probably be recoded as:

```
#define A b /* comment */ \
```

692 **decimal character '*string*' follows octal escape sequence '*string*'**
warning A *string* was found that contains an '8' or '9' after an octal escape sequence with no more than two octal digits, e.g.

```
"\079"
```

contains two characters: Octal seven (ASCII BEL) followed by '9'. The casual reader of the code (and perhaps even the programmer) could be fooled into thinking this is a single character. If this is what the programmer intended he can also render this as

```
"\07" "9"
```

so that there can be no misunderstanding. On the other hand,

```
"\1238"
```

will not raise a message because it is assumed that the programmer knows that octal escape sequences cannot exceed four characters (including the initial backslash).

693 **the sequence "*detail*" represents a NUL character followed by the literal string "*detail*"**
warning A string was found that looks suspiciously like (but is not) a hexadecimal escape sequence; rather, it is a null character followed by letter `"x"` followed by some hexadecimal digit, e.g.:

```
"\0x62"
```

was found where the programmer probably meant to type `"\x62"`. If you need precisely this sequence you can use:

```
"\0" "x62"
```

and this warning will not be issued.

696 **values from '*integer*' to '*integer*' are out of range for operator '*string*'**
warning The variable is being compared (using one of the 6 comparison operations) with some other expression called the comperand. The variable has a value that is out of the range of values of this comperand. For example consider:

```
void f(unsigned char ch) {
    int n = 1000;
    if (ch < n)     // Message 696
    ...
}
```

Here a message `696` will be issued stating that `n` has a value of `1000` that is out of range because `1000` is not in the set of values that `ch` can hold (assuming default sizes of scalars).

697 **an expression with an integral strong boolean type should be equality-compared only to zero**
warning A quasi-boolean value is being compared (using either `!=` or `==`) with a value that is not the literal zero. A quasi-boolean value is any value whose type is a strong boolean type and that could conceivably be something other than zero or one. This is significant because in C, all non-zero values are equally true. Example:

```
/*lint -strong(AJXb, B) */
typedef int B;
#define YES ((B)1)
#define NO ((B)0)

B f( B a, B b ) {
    B c = ( a == NO);              /*OK, no Warning here*/
    B d = ( a == (b != NO) );   /* Warning 697 for == but not for != */
    B e = ( a == YES );         /* Warning 697 here */
    return d == c;              /* Warning 697 here */
}
```

Note that if `a` and `b` had instead been declared with true boolean types, such as '`bool`' in C++ or '`_Bool`' in C99, this diagnostic would not have been issued.

698 **in-place realloc of *symbol* could cause a memory leak**
warning A statement of the form:

```
v = realloc( v, ... );
```

has been detected. Note the repeated use of the same variable. The problem is that `realloc` can fail to allocate the necessary storage. In so doing it will return NULL. But then the original value of `v` is overwritten resulting in a memory leak.

701 **shift left of signed quantity (*type*)**
info Shifts are normally accomplished on `unsigned` operands.

702 **shift right of signed quantity (*type*)**
info Shifts are normally accomplished on `unsigned` operands. Shifting an `int` right is machine dependent (sign fill vs. zero fill).

703 **shift left of signed quantity (*type*)**
info Shifts are normally accomplished on `unsigned` operands.

704 **shift right of signed quantity (*type*)**
info Shifts are normally accomplished on `unsigned` operands. Shifting a `long` to the right is machine dependent (sign fill vs. zero fill).

705 **format '*string*' specifies type *type* which is nominally inconsistent with argument no. *integer* of *string type***
info The argument corresponding to a conversion specifier in a `printf`/`scanf` style function was not of the correct type but was the same size as the expected integer type. The format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
extern char * buffer;
sprintf(buffer, "%u", 371);
```

will elicit the message:

```
format '%u' specifies type 'unsigned int' which is nominally
inconsistent with argument no. 3 of type 'int'
```

In addition to differences in signedness of same-sized integers, two types that are the same size and signedness but distinct types are also reported by this message. For example, if `int` and `long` are the same size, passing a `long` argument to `%d` will elicit this message.

706 info **format '*string*' specifies type *type* whose pointee type is nominally inconsistent with argument no. *integer* of *string type***

The argument corresponding to a conversion specifier in a `printf`/`scanf` style function was not of the correct type but was a pointer to a type that is the same size as the expected pointee integer type. The format, expected argument type, argument number, and the type of the data argument provided are reported. Argument counts begin at 1 and include file, string, and data arguments. For example:

```
int j;
scanf("%u", &j);
```

will result in the message:

```
format '%u' specifies type 'unsigned int *' whose pointee type
is nominally inconsistent with argument no. 2 of type 'int *'
```

In addition to differences in signedness of same-sized integers, pointers to types that are the same size and signedness but distinct types are also reported by this message. For example, if `int` and `long` are the same size, passing a `long *` argument to `%d` will elicit this message.

707 info **mixing narrow and wide string literals in concatenation**

The following is an example of a mixing of narrow and wide string literals.

```
const wchar\_t *s = "abc" L"def";
```

The concatenation of narrow and wide string literals results in undefined behavior for C90 and C++2003. If your compiler supports such combinations or you use a C/C++ dialect that supports such, you may either suppress this message or consider making the concatenands match.

Supports MISRA C++ Rule 2-13-5 (Req)

708 info **union initialization**

A union was initialized without explicitly specifying which member to initialize. While the C and C++ standards state that the first member of the union is initialized in such cases, other members may not have fully initialized values. For example:

```
union U { int a; int * p; };
U u1 = { 0 };
```

On a system where `int` is 4 bytes and pointers are 8 bytes, the `int` member of `u1` is initialized to 0 but the bytes of `p` that do not overlap with `a` are not initialized, which may come as a surprise, especially since the behavior is dependent on the order in which the union members are declared and on the size of pointers relative to `int`s.

709 info **no intervening module since the last '-pch' option**

Two `-pch` options were seen without an intervening module. This is suspicious because the first `-pch` option has no effect in such a case as only one PCH file can be used per module.

712 info **implicit conversion (*context*) from *type* to *type***

An assignment (or implied assignment, see *context*) is being made from a source *type* (the first *type*) to a destination type (the second *type*) and the first *type* is larger than the second *type*. A cast will suppress this message.

713 implicit conversion (*context*) from *type* to *type*
info An assignment (or implied assignment, see *context*) is being made from an unsigned quantity to a signed quantity, that will result in the possible loss of one bit of integral precision, such as converting from `unsigned int` to `int`. A cast will suppress the message.

714 external symbol *symbol* was defined but not referenced
info The named external variable or external function was defined but not referenced. This message is suppressed for unit checkout (`-unit_check` option).
Supports MISRA C++ Rule 0-1-3 (Req)
Supports MISRA C++ Rule 0-1-10 (Req)

715 named parameter *symbol* of *symbol* not subsequently referenced
info The named formal parameter was not referenced.
Supports MISRA C 2012 Rule 2.7 (Adv)
Supports MISRA C++ Rule 0-1-11 (Req)
Supports MISRA C++ Rule 0-1-12 (Req)

716 infinite loop via while
info 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 monocarpic do-while used to group statements
info 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. [6, Section 19.7] For example:

```
#define f(k) do {n=k; m=n+1;} while(0)
```

allows `f(k)` to be used in conditional statements as in

```
if(n>0) f(3);
else f(2);
```

Thus, if you are doing this deliberately use `-e717`

718 function *symbol* undeclared, assumed to return int
info A function was referenced without having been declared or defined within the current module. Such implicit function declarations were removed in C99 although some compilers still allow them. These implicit function declarations were never allowed in C++ and referencing an undeclared function in a C++ module will instead result in an error. Note that by adding a declaration to another module, you will not suppress this message. It can only be suppressed by placing a declaration within the module being processed.
Supports MISRA C 2004 Rule 8.1 (Req)
Supports MISRA C 2012 Rule 17.3 (Mand)

719 data argument *integer* not used by format string
info The number of data arguments passed to a `printf/scanf` style function was more than what is specified in the format. This message is similar to Warning 558, which alerts users to situations in which there were

too few arguments for the format. It receives a lighter Informational classification because the additional arguments are simply ignored whereas passing too few arguments results in undefined behavior.

720 boolean test of assignment
info
An assignment was found in a context that requires a Boolean (such as the condition of an `if` or `while` statement). This may be legitimate or it could have resulted from a mistaken use of `=` for `==`. If the assignment was intentional, placing additional parenthesis around the assignment (e.g. `if ((a = b))`) will suppress this message.
Supports MISRA C 2004 Rule 13.1 (Req)
Supports MISRA C 2012 Rule 13.4 (Adv)
Supports MISRA C++ Rule 6-2-1 (Req)

721 if statement has empty body
info
A semi-colon was found immediately to the right of a right parenthesis in a construct of the form `if(`*e*`);`. As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the '`;`' is separated by at least one blank from the '`)`'. Better, place it on a separate line. See also message 548.

722 '*context*' statement has empty body
info
A semi-colon was found immediately to the right of a right parenthesis in a construct of the form `while(`*e*`);` or `for(`*e*`;`*e*`;`*e*`);` As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the '`;`' is separated by at least one blank from the '`)`'. Better, place it on a separate line.

723 macro definition starting with = is suspicious
info
A preprocessor definition began with an `=` sign. For example:

```
#define LIMIT = 50
```

Was this intentional? Or was the programmer thinking of assignment when he wrote this?

725 unexpected lack of indentation
warning
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 11.4 Indentation Checking).

726 extraneous comma ignored at end of enumerator list after enumerator *symbol*
info
A comma followed by a right-brace within an enumeration is not a valid ANSI/ISO construct. The comma is ignored. This message is only emitted in C90 and C++03 modes as later versions allow this construct.

727 static local symbol *symbol* not explicitly initialized
info
The named static variable (local to a function) was not explicitly initialized before use. The following remarks apply to messages 728 and 729 as well as 727. By no explicit initialization we mean that there was no initializer present in the definition of the object, no direct assignment to the object (or any of its elements or members), and no address operator applied to the object or, if the address of the object was taken, it was assigned to a pointer to const. Arrays are also considered to be explicitly initialized if the result of array to

pointer decay is assigned to a non-const pointer.

These messages do not necessarily signal errors since the implicit initialization for static variables is 0. However, the messages are helpful in indicating those variables that you had forgotten to initialize to a value. To extract the maximum benefit from the messages we suggest that you employ an explicit initializer for those variables that you want to initialize to 0. For example:

```
static int n = 0;
```

For variables that will be initialized dynamically, do not use an explicit initializer as in:

```
static int m;
```

This message will be given for any array, `struct` or `union` if no member or element has been assigned a value.

728 file scope static variable *symbol* not explicitly initialized
info
The named intra-module variable (static variable with file scope) was not explicitly initialized. See the comments on message 727 for more details.

729 external variable *symbol* not explicitly initialized
info
The named inter-module variable (external variable) was not explicitly initialized. See the comments on message 727 for more details. This message is suppressed for unit checkout (`-unit_check`).

730 boolean used as argument *integer* to function *symbol*
info
A Boolean was used as an argument to a function. Was this intended? Or was the programmer confused by a particularly complex conditional statement? Experienced C programmers often suppress this message. This message is given only if the associated parameter is not declared bool.

731 boolean argument(s) to *equality-operator*
info
A Boolean operator was used as an argument to `==` or `!=`. For example:

```
if( (a > b) == (c > d) )          ...
```

tests to see if the inequalities are of the same value. This could be an error as it is an unusual use of a Boolean (see Warnings 503 and 514) but it may also be deliberate since this is the only way to efficiently achieve equivalence or exclusive or. Because of this possible use, the construct is given a relatively mild 'informational' classification. If the Boolean argument is cast to some type, this message is not given. Additionally, this message is not necessarily given just because one of the arguments to `==` or `!=` is a Boolean type but only if at least one of the arguments is expressed using a Boolean operator. For example, if `e` and `f` are of type bool, the clause:

```
if( e == f ) ...
```

will not prompt this message. However,

```
if( e == !f ) ...
```

will.

732 loss of sign (*context*) (*type* to *type*)
info
An assignment (or implied assignment, see *context*) is made from a signed quantity to an unsigned quantity. Also, it could not be determined that the signed quantity had no sign. For example:

```
u = n;     /* Info 732 */
u = 4;     /*  OK      */
```

where `u` is unsigned and `n` is not, warrants a message only for the first assignment, even though the constant `4` is nominally a signed `int`.

Make sure that this is not an error (that the assigned value is never negative) and then use a cast (to `unsigned`) to remove the message.

733 likely assigning address of local *symbol* to outer scope pointer *symbol*
info The address of an `auto` variable is valid only within the block in which the variable is declared. An address to such a variable has been assigned to a variable that has a longer life expectancy. There is an inherent danger in doing this.
Supports MISRA C 2004 Rule 17.6 (Req)
Supports MISRA C 2012 Rule 18.6 (Req)

734 loss of precision (*context*) from *number* bits to *number* bits
info An assignment is being made into an object smaller than an `int`. The information being assigned is derived from another object or combination of objects in such a way that information could potentially be lost. The number of bits given does not count the sign bit. For example if `ch` is a `char` and `n` is an `int` then:

```
ch = n;
```

will trigger this message whereas:

```
ch = n & 1;
```

will not. To suppress the message a cast can be made as in:

```
ch = (char) n;
```

You may receive notices involving multiplication and shift operators with subinteger variables. For example:

```
ch = ch << 2
ch = ch * ch
```

where, for example, `ch` is an `unsigned char`. These can be suppressed by using the flag `+fpm` (precision of an operator is bound by the maximum of its operands).

735 implicit conversion (*context*) from *type* to *type*
info An assignment (or implied assignment, see *context*) is made from a `long double` to a `double`. Using a cast will suppress the message. The number of bits includes the sign bit.

736 loss of precision (*context*) from *number* bits to *number* bits
info An assignment (or implied assignment, see *context*) is being made to a `float` from a value or combination of values that appear to have higher precision than a `float`. You may suppress this message by using a cast. The number of bits includes the sign bit.

737 loss of sign in promotion from *type* to *type*
info An unsigned quantity was joined with a signed quantity in a binary operator (or 2nd and 3rd arguments to the conditional operator) and the signed quantity is implicitly converted to `unsigned`. The message will not be given if the signed quantity is an unsigned constant, a Boolean, or an expression involving bit manipulation. For example,

```
u & ~0xFF
```

where `u` is unsigned does not draw the message even though the operand on the right is technically a signed integer constant. It looks enough like an unsigned to warrant not giving the message.

This mixed mode operation could also draw Warnings 573 or 574 depending upon the operator involved.

You may suppress the message with a cast but you should first determine whether the signed value could ever be negative or whether the unsigned value can fit within the constraints of a signed quantity.

738 address of static local symbol *symbol* not explicitly initialized before passed to a function
info
The named static local variable was not initialized before being passed to a function whose corresponding parameter is declared as pointer to `const`. Is this an error or is the programmer relying on the default initialization of 0 for all static items? By employing an explicit initializer you will suppress this message. See also message numbers 727 and 603.

739 trigraph sequence '*string*' in string literal
info
The indicated Trigraph (three-character) sequence was found within a string. This trigraph reduces to a single character according to the ANSI/ISO standards. This represents a "Quiet Change" from the past where the sequence was not treated as exceptional. If you had no intention of mapping these characters into a single character you may precede the initial '`?`' with a backslash. If you are aware of the convention and you intend that the Trigraph be converted you should suppress this informational message.
Supports MISRA C 2004 Rule 4.2 (Req)
Supports MISRA C 2012 Rule 4.2 (Adv)
Supports MISRA C++ Rule 2-3-1 (Req)

742 multi-character character constant
info
A character constant was found that contained multiple characters, e.g., '`ab`'. This is legal C but the numeric value of the constant is implementation defined. It may be safe to suppress this message because, if more characters are provided than what can fit in an `int`, message number 25 is given.

743 negative character constant
info
A character constant was specified whose value is some negative integer. For example, on machines where a byte is 8 bits, the character constant '`\xFF`' is flagged because its value (according to the ANSI/ISO standard) is `-1` (its type is `int`). Note that its value is not `0xFF`.

744 switch statement has no default
info
A `switch` statement has no section labeled `default`. Was this an oversight? It is standard practice in many programming groups to always have a `default:` case. This can lead to better (and earlier) error detection. One way to suppress this message is by introducing a vacuous `default break;` statement. If you think this adds too much overhead to your program, think again. In all cases tested so far, the introduction of this statement added absolutely nothing to the overall length of code. If you accompany the vacuous statement with a suitable comment, your code will at least be more readable.

This message is not given if the control expression is an enumerated type. In this case, all enumerated constants are expected to be represented by `case` statements, else 787 will be issued.
Supports MISRA C 2012 Rule 16.4 (Req)
Supports MISRA C++ Rule 6-4-6 (Req)

746 call to function *symbol* not made in the presence of a prototype
info

A call to a function is not made in the presence of a prototype. This does not mean that PC-lint Plus is unaware of any prototype; it means that a prototype is not in a position where a compiler will see it. If you have not adopted a strict prototyping convention you will want to suppress this message with `-e746`.
Supports MISRA C 2004 Rule 8.1 (Req)

749 local enumeration constant *symbol* not referenced
info A member (name provided as *symbol*) of an `enum` was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. Compare with messages `754` and `769`.

750 local macro '*string*' not referenced
info A 'local' macro is one that is not defined in a header file. The macro is not referenced throughout the module in which it is defined.
Supports MISRA C 2012 Rule 2.5 (Adv)

751 local typedef *symbol* not referenced
info A 'local' `typedef` symbol is one that is not defined in any header file. It may have file scope or block scope but it was not used through its scope.
Supports MISRA C 2012 Rule 2.3 (Adv)
Supports MISRA C++ Rule 0-1-5 (Req)

752 local declarator *symbol* not referenced
info A 'local' declarator symbol is one declared in a declaration appearing in the module file itself as opposed to a header file. The symbol may have file scope or may have block scope. But it wasn't referenced.
Supports MISRA C++ Rule 0-1-3 (Req)

753 local *string* *symbol* not referenced
info *string* is one of `struct`, `class`, `union`, or `enum` and *symbol* is the name of the tag. A 'local' tag is one not defined in a header file. Since its definition appeared, why was it not used? Use of a tag is implied by the use of any of its members.
Supports MISRA C 2012 Rule 2.4 (Adv)
Supports MISRA C++ Rule 0-1-5 (Req)

754 local *string* member *symbol* not referenced
info A member (name provided as *symbol*) of a `struct`, `class`, or `union` (as indicated in *string*) was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. See message `768`.

755 global macro '*string*' not referenced
info A 'global' macro is one defined in a header file. The macro is not used in any of the modules comprising the program. This message is suppressed for unit checkout (`-unit_check` option).
Supports MISRA C 2012 Rule 2.5 (Adv)

756 global typedef *symbol* not referenced
info This message is given for a `typedef` symbol declared in a non-library header file. The symbol is not used in any of the modules comprising a program. This message is suppressed for unit checkout (`-unit_check`

option).
Supports MISRA C 2012 Rule 2.3 (Adv)
Supports MISRA C++ Rule 0-1-5 (Req)

757 global declarator *symbol* not referenced
info 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 (`-unit_check`).
Supports MISRA C++ Rule 0-1-3 (Req)

758 global *string symbol* not referenced
info This message is given for `struct`, `union` and `enum` tags that have been defined in non-library header files and that have not been used in any module comprising the program. The message is suppressed for unit checkout (`-unit_check`).
Supports MISRA C++ Rule 0-1-5 (Req)

759 header declaration for symbol *symbol* could be moved from header to module
info 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 given only when more than one module is being linted.
Supports MISRA C++ Rule 3-3-1 (Req)

760 redundant macro *name* defined identically
info The given macro was defined earlier in the same way and is hence redundant.
Supports MISRA C 2012 Rule 5.4 (Req)

761 redundant typedef *name*
info A `typedef` symbol has been declared earlier and is redundant. Although the declarations are consistent you should probably remove the second.

764 switch with no cases
info A `switch` statement has been found that does not have a `case` statement associated with it (it may or may not have a `default` statement). This is normally a useless construct.
Supports MISRA C 2004 Rule 15.5 (Req)
Supports MISRA C 2012 Rule 16.6 (Req)
Supports MISRA C++ Rule 6-4-8 (Req)

765 external symbol *symbol* could be made static
info An external symbol was referenced in only one module. It was not declared `static`. Some programmers like to make `static` every symbol they can, because this lightens the load on the linker. It also represents good documentation.
Supports MISRA C 2004 Rule 8.10(Req)
Supports MISRA C 2012 Rule 8.7 (Adv)
Supports MISRA C++ Rule 3-3-1 (Req)

767 macro '*string*' was defined differently in another module
info Two macros processed in two different modules had inconsistent definitions.

768 global structure member *symbol* not referenced
info A member (name provided as *symbol*) of a `struct` or `union` appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. Since a `struct` may be replicated in storage, finding an unused member can pay handsome storage dividends. However, many structures merely reflect an agreed upon convention for accessing storage and for any one program, many members are unused. In this case, receiving this message can be a nuisance. One convenient way to avoid unwanted messages (other than the usual `-e` and `-esym`) is to always place such structures in library header files. Alternatively, you can place the struct within a `++flb` ... `--flb` sandwich to force it to be considered library.

769 global enumeration constant *symbol* not referenced
info A member (name provided as *symbol*) of an `enum` appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. There are reasons why a programmer may occasionally want to retain an unused `enum` and for this reason this message is distinguished from `768` (unused member). See message `768` for ways of selectively suppressing this message.

770 tag *symbol* defined identically at *location*
info The `struct`, `union`, or `enum` tag *symbol* was defined identically in different locations (usually two different files). This is not an error but it is not necessarily good programming practice either. It is better to place common definitions of this kind in a header file where they can be shared among several modules. If you do this, you will not get this message. Note that if the tag is defined differently in different scopes, you will receive warning `631` rather than this message.

773 expression-like macro '*string*' not parenthesized
info A macro that appeared to be an expression contained unparenthesized binary operators and therefore may result in unexpected associations when used with other operators. For example,

```
#define A B + 1
```

may be used later in the context:

```
f( A * 2 );
```

with the surprising result that `B+2` gets passed to `f` and not the `(B+1)*2`. Corrective action is to define `A` as:

```
#define A (B + 1)
```

Highest precedence binary operators are not reported upon. Thus:

```
#define A s.x
```

does not elicit this message because this case does not seem to represent a problem. Also, unparenthesized unary operators (including casts) do not generate this message. [6, Section 19.5]

774 boolean condition for '*detail*' always evaluates to '*detail*'
info The indicated clause (*detail* is one of `if`, `while` or `for` (2nd expression)) has an argument that appears to always evaluate to either '`true`' or '`false`' (as indicated in the message). Information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message `506`, which is based on testing constants or combinations of constants. Also compare with the Elective Note `944`, which can sometimes provide more detailed information. See Chapter 8 Value Tracking.

Supports MISRA C 2004 Rule 13.7 (Req)
Supports MISRA C 2012 Rule 14.3 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)
Supports MISRA C++ Rule 0-1-2 (Req)
Supports MISRA C++ Rule 0-1-9 (Req)

775 nonnegative quantity cannot be less than zero
info 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
info An `int` expression (signed or unsigned) involving addition or subtraction is converted to `long` implicitly or explicitly. Moreover, the precision of a `long` is greater than that of `int`. If an overflow occurred, information would be lost. Either cast one of the operands to some form of `long` or cast the result to some form of `int`.

See Warning 647 for a further description and an example of this kind of error. See also messages 790 and 942.

777 testing floating point values for equality
info This message is issued when the operands of operators `==` and `!=` are some form of floating type (`float`, `double`, or `long double`). Testing for equality between two floating point quantities is suspect because of round-off error and the lack of perfect representation of fractions. If your numerical algorithm calls for such testing turn the message off. The message is suppressed when one of the operands can be represented exactly, such as 0 or 13.5.
Supports MISRA C 2004 Rule 13.3 (Req)

778 constant expression evaluates to 0 in '*unary/binary*' operation '*operator*'
info A constant expression involving addition, subtraction, multiplication, shifting, or negation resulted in a 0. This could be a purposeful computation but could also have been unintended. If this is intentional, suppress the message. If one of the operands is 0, Elective Note 941 may be issued rather than a 778.

779 string constant in comparison operator '*operator*'
info A string constant appeared as an argument to a comparison operator. For example:

```
if( s == "abc" ) ...
```

This is usually an error. Did the programmer intend to use `strcmp`? It certainly looks suspicious. At the very least, any such comparison is bound to be machine-dependent. If you cast the string constant, the message is suppressed.

783 line does not end with a newline
info This message is issued when an input line is not terminated by a new-line or when a NUL character appears within an input line. If your editor is in the habit of not appending new-lines onto the end of the last line of the file then suppress this message. Otherwise, examine the file for NUL characters and eliminate them.

784 nul character truncated from string
info
During initialization of an array with a string constant there was not enough room to hold the trailing NUL character. For example:

```
char a[3] = "abc";
```

would evoke such a message. This may not be an error since the easiest way to do this initialization is in the manner indicated. It is more convenient than:

```
char a[3] = { 'a', 'b', 'c' };
```

On the other hand, if it really is an error it may be especially difficult to find.

785 too few initializers for aggregate of type *type* in initialization of *symbol*
info
The number of initializers in a brace-enclosed initializer was less than the number of items in the aggregate. Default initialization is taken. An exception is made with the initializer `{0}`. This is given a separate message number in the Elective Note category (943). It is normally considered to be simply a stylized way of initializing all members to 0. See also 9068.

786 string concatenation within initializer
info
Although it is perfectly 'legal' to concatenate string constants within an initializer, this is a frequent source of error. Consider:

```
char *s[] = { "abc" "def" };
```

Did the programmer intend to have an array of two strings but forget the comma separator? Or was a single string intended?

787 enum constant *symbol* not used within switch
info
A `switch` expression is an enumerated type and at least one of the enumerated constants was not present as a `case` label. Moreover, no `default` case was provided.

788 enum constant *symbol* not used within default switch
info
A `switch` expression is an enumerated type and at least one of the enumerated constants was not present as a `case` label. However, unlike Info `787`, a `default` case was provided. This is a mild form of the case reported by Info `787`. The user may thus elect to inhibit this mild form while retaining Info `787`.

789 assigning address of auto variable *symbol* to static
info
The address of an `auto` variable (*symbol*) is being assigned to a `static` variable. This is dangerous because the `static` variable will persist after return from the function in which the `auto` is declared but the `auto` will be, in theory, gone. This can prove to be among the hardest bugs to find. If you have one of these, make certain there is no error and use `-esym` to suppress the message for a particular variable.
Supports MISRA C 2004 Rule 17.6 (Req)
Supports MISRA C 2012 Rule 18.6 (Req)
Supports MISRA C++ Rule 7-5-2 (Req)

790 possibly truncated *string* promoted to *type*
info
This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals, the result of which is later converted to a floating point quantity. The operations that are scrutinized and reported upon by this message are: shift left and multiplication. Addition and subtraction are covered by note 942. See also messages 647 and 776.

791 a single line suppression followed a normal option
info A temporary message suppression option (one having the form: `!e...`) followed a regular option. Was this intended?

792 casting void expression to void
info A `void` expression has been cast to `void`. Was this intended?

793 ANSI/ISO minimum translation limit of *integer* '*string*' exceeded, processing is unaffected
info An ANSI/ISO minimum translation limit has been exceeded. These limits are described under the heading "Translation limits" in the ANSI/ISO C Standards and under the heading "Implementation Quantities" in the C++ standards. Programs exceeding these limits are not considered maximally portable. However, they may work for individual compilers.

The *integer* parameter indicates the numeric value that was exceeded and *string* provides a textual description of the limit in question.

Say a large program exceeds the ANSI/ISO limit of 4095 external identifiers. This will result in the message:

```
793 ANSI/ISO minimum translation limit of 4095 'external identifiers'
    exceeded, processing is unaffected
```

It may not be obvious how to inhibit this message for identifiers while leaving other limits in a reportable state. The second parameter of the message is designated *string* and so the `-estring` may be used. Because the string contains a blank, quotes must be used. The option becomes:

```
-estring(793,"external identifiers')
```

See 11.10 Language Limits for additional information and a list of supported limits.

798 redundant char '*character*'
info The indicated character is redundant and can be eliminated from the input source. A typical example is a backslash on a line by itself.

799 numerical constant '*integer*' larger than unsigned long
info An integral constant was found to be larger than the largest value allowed for `unsigned long` quantities. By default, an `unsigned long` is 4 bytes but can be respecified via the option `-sl#`. If the `long long` type is permitted (see option `+fll`) this message is automatically suppressed. See also message 417.

801 goto statement used
info A `goto` was detected. Use of the `goto` is not considered good programming practice by most authors and its use is normally discouraged. There are a few cases where the `goto` can be effectively employed but often these can be rewritten just as effectively without the `goto`. The use of `goto` statements can have a devastating effect on the structure of large functions creating a mass of spaghetti-like confusion. For this reason its use has been banned in many venues.
Supports MISRA C 2004 Rule 14.4 (Req)
Supports MISRA C 2012 Rule 15.1 (Adv)

805 expected L"..." to initialize wide char string
info An initializer for a wide character array or pointer did not use a preceding 'L'. For example:

```
wchar_t a[] = "abc";
```

was found whereas

```
wchar_t a[] = L"abc":
```

was expected.

806 small signed bitfield
info
A small bit field (less than an `int` wide) was found and the base type is signed rather than unsigned. Since the most significant bit is a sign bit, this practice can produce surprising results. For example,

```
struct { int b:1; } s;
s.b = 1;
if( s.b > 0 ) /* should succeed but actually fails */
    ...
```

808 no explicit type given, int assumed
info
An explicit type was missing in a declaration. Unlike Warning 601, the declaration may have been accompanied by a storage class or modifier (qualifier) or both. For example:

```
extern f(void);
```

will draw message 808. Had the `extern` not been present, a **745** would have been raised.

The keywords `unsigned`, `signed`, `short` and `long` are taken to be explicit type specifiers even though `int` is implicitly assumed as a base.
Supports MISRA C 2004 Rule 8.2 (Req)
Supports MISRA C 2012 Rule 8.1 (Req)

809 likely returning address of local *symbol* through *symbol*
info
The value held by a pointer variable may have been the address of an `auto` variable. It is normally incorrect to return the address of an item on the stack because the portion of the stack allocated to the returning function is subject to being obliterated after return.

810 arithmetic modification of custodial pointer
info
We define the custodial variable as that variable directly receiving the result of a `malloc` or `new` or equivalent call. It is inappropriate to modify such a variable because it must ultimately be `free`'ed or `delete`'ed. You should first make a copy of the custodial pointer and then modify the copy. The copy is known as an alias.

812 static variable *symbol* is *integer* bytes
info
The amount of storage for a static symbol has reached or exceeded a value that was specified in a `-size` option.

813 auto variable *symbol* is *integer* bytes
info
The amount of storage for an auto symbol has reached or exceeded a value that was specified in a `-size` option.

814 tagless struct without a declarator is useless here
info
A tagless struct was declared without a declarator. For example:

```
struct { int n; };
```

Such a declaration cannot very well be used.

815 unsaved pointer used in pointer arithmetic
info
An allocation expression (`malloc`, `calloc`, `new`) is not immediately assigned to a variable but is used as an operand in some expression. This would make it difficult to free the allocated storage. For example:

```
p = new X[n] + 2;
```

will elicit this message. A preferred sequence is:

```
q = new X[n];
p = q+2;
```

In this way the storage may be freed via the custodial pointer `q`.

Another example of a statement that will yield this message is:

```
p = new (char *) [n];
```

This is a gruesome blunder on the part of the programmer. It does NOT allocate an array of pointers as a novice might think. It is parsed as:

```
p = (new (char *)) [n];
```

which represents an allocation of a single pointer followed by an index into this 'array' of one pointer.

816 non-ISO format specification '*string*'
info
A non-standard format specifier was found in a format-processing function such as `printf` or `scanf`. The format was recognized as being a common extension. If the format was not recognized, a more severe warning (557) would have been issued. The non-ISO conversion specifiers that are recognized are:

`%C`	`wchar_t`	XSI/MS extension, equivalent to `%lc`
`%D`	`int`	Apple extension, synonym for `%d`
`%O`	`unsigned int`	Apple extension, synonym for `%o`
`%S`	`wchar_t *`	XSI/MS extension, equivalent to `%ls`
`%U`	`unsigned int`	Apple extension, synonym for `%u`
`%Z`	`ANSI_STRING` / `UNICODE_STRING`	MS extension
`%m`	`none`	Glibc extension, prints output of `strerror(errno)`

818 parameter *symbol* of function *symbol* could be pointer to const
info
As an example:

```
int f( int *p ) { return *p; }
```

can be redeclared as:

```
int f( const int *p ) { return *p; }
```

Declaring a parameter a pointer to `const` offers advantages that a mere pointer does not. In particular, you can pass to such a parameter the address of a `const` data item. In addition it can offer better documentation.

Other situations in which a `const` can be added to a declaration are covered in messages 952, 953, 954 and 1764.

Supports MISRA C 2004 Rule 16.7 (Adv)
Supports MISRA C 2012 Rule 8.13 (Adv)
Supports MISRA C++ Rule 7-1-2 (Req)

820 **boolean test of parenthesized assignment**
info A Boolean test was made on the result of an assignment and, moreover, the assignment was parenthesized. For example:

```
if ( (a = b) ) ...  // Info 820
```

will draw this informational whereas

```
if ( a = b )  ...  // Info 720
```

(i.e. the unparenthesized case) will, instead, draw Info 720. We, of course, do not count the outer parentheses, required by the language, that always accompany the `if` clause.

The reason for partitioning the messages in this fashion is to allow the programmer to adopt the convention, advanced by some compilers (in particular `gcc`), of always placing a redundant set of parentheses around any assignment that is to be tested. In this case you can suppress Info 820 (via `-e820`) while still enabling Info 720.

Supports MISRA C 2012 Rule 13.4 (Adv)
Supports MISRA C++ Rule 6-2-1 (Req)

821 **right hand side of assignment not parenthesized**
info An assignment operator was found having one of the following forms:

```
a = b || c
a = b && c
a = b ? c : d
```

Moreover, the assignment appeared in a context where a value was being obtained. For example:

```
f( a = b ? c : d );
```

The reader of such code could easily confuse the assignment for a test for equality. To eliminate any such doubts we suggest parenthesizing the right hand side as in:

```
f( a = (b ? c : d) );
```

825 **control flow falls through to next case without an intervening -fallthrough comment**
info A common programming mistake is to forget a break statement between case statements of a switch. For example:

```
case 'a': a = 0;
case 'b': a++;
```

Is the fall through deliberate or is this a bug? To signal that this is intentional use the `-fallthrough` option within a lint comment as in:

```
case 'a': a = 0;
    //lint -fallthrough
case 'b':  a++;
```

This message is similar to Warning 616 ("control flows into case/default") and is intended to provide a stricter alternative. Warning 616 is suppressed by any comment appearing at the point of the fallthrough. Thus, an accidental omission of a break can go undetected by the insertion of a neutral comment. This can be hazardous to well-commented programs.

826 **suspicious pointer-to-pointer conversion (area too small)**
info A pointer was converted into another either implicitly or explicitly. The area pointed to by the destination pointer is larger than the area that was designated by the source pointer. For example:

```
long *f( char *p ) { return (long *) p; }
```

827 **loop can only be reached via goto due to unconditional transfer of control by '*string*' statement**
info
A loop structure (`for`, `while`, or `do`) could not be reached. Was this an oversight? It may be that the body of the loop has a labeled statement and that the plan of the programmer is to jump into the middle of the loop through that label. It is for this reason that we give an Informational message and not the Warning (527) that we would normally deliver for an unreachable statement. But please note that jumping into a loop is a questionable practice in any regard.
Supports MISRA C 2004 Rule 14.1 (Req)
Supports MISRA C 2012 Rule 2.1 (Req)
Supports MISRA C++ Rule 0-1-1 (Req)
Supports MISRA C++ Rule 0-1-2 (Req)

828 **semantics of '*string*' copied to function '*string*'**
info
A function with built-in semantics or user-defined semantics was `#define`'d to be some other function with a similar name formed by prepending or appending underscores. For example:

```
#define strcmp(a,b) __strcmp__(a,b)
```

will cause Info 828 to be issued. As the message indicates, the semantics will be automatically transferred to the new function.

829 **a +headerwarn option was previously issued for header '*file*'**
info
Some coding standards discourage or even prohibit the use of certain header files. PC-lint Plus can guard against their use by activating the lint option `+headerwarn`(*file*). Later, if the file is used, we will then issue this message.
Supports MISRA C 2004 Rule 20.8 (Req)
Supports MISRA C 2004 Rule 20.9 (Req)
Supports MISRA C 2012 Rule 17.1 (Req)
Supports MISRA C 2012 Rule 21.4 (Req)
Supports MISRA C 2012 Rule 21.5 (Req)
Supports MISRA C 2012 Rule 21.10 (Req)
Supports MISRA C 2012 Rule 21.11 (Req)
Supports MISRA C++ Rule 18-0-1 (Req)
Supports MISRA C++ Rule 18-0-4 (Req)
Supports MISRA C++ Rule 18-7-1 (Req)
Supports MISRA C++ Rule 27-0-1 (Req)

831 **value tracking history *text varies***
supplemental
This message provides supplemental value tracking history information and is attached to a proceeding value tracking message. Multiple `831` messages may be provided for single value tracking message and the exact verbiage will vary depending on the situation. For example:

```
short f(short x, short y) {
    if (x >= 10 && x <= 20) {
        return y / (x - 15);
    }
    ....
```

results in:

```
warning 414: possible division by zero
```

```
        return y / (x - 15);
                 ^ ~~~~~~~~
supplemental 831: operator - yields -5:5
        return y / (x - 15);
                    ~~^~~~
supplemental 831: integral conversion yields 10:20
        return y / (x - 15);
                    ^
supplemental 831: inference yields 10:20
    if (x >= 10 && x <= 20) {
    ^                ~~~~~~~
supplemental 831: inference yields 10:32767
    if (x >= 10 && x <= 20) {
    ^   ~~~~~~~
```

The main message is 414 - "possible division by zero" and the 831 messages show the steps that lead PC-lint Plus to make this determination. Starting from the bottom up, the last message indicates the left-hand side of the && operator resulted in an inference that the lower bound of x must be 10. The right-hand side served to constrain the upper bound of x. At the point of the return statement, PC-lint Plus knows that the value of x is between 10 and 20, inclusive. The value of the expression x - 15 therefore is between -5 and 5. Since this is a constrained range that contains the value zero and is being used as a divisor, message 414 is issued.

832 info **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.

Supports MISRA C 2012 Rule 8.1 (Req)

834 info **operator '*operator*' followed by operator '*operator*' could be confusing without parentheses**

Some combinations of operators seem to be confusing. For example:

```
a = b - c - d;
a = b - c + d;
a = b / c / d;
a = b / c * d;
```

tend to befuddle the reader. To reduce confusion we recommend using parentheses to make the association of these operators explicit. For example:

```
a = (b - c) - d;
a = (b - c) + d;
a = (b / c) / d;
a = (b / c) * d;
```

in place of the above.

835 info **zero given as *string* argument to *operator context***

A 0 has been provided as an operand to an arithmetic operator. The name of the operator is provided in the message as well as the side of the operator (left or right) that had the unusual value. For example:

```
n = n + 0 - m;
```

will produce a message that the right hand operand of `operator '+'` is zero. In general the operators examined are the binary operators:

```
+ - * / % | & ^ << >>
```

and the unary operators `-` and `+`. An enumeration constant whose value is 0 is permitted with operators:

```
+ - >> <<
```

Otherwise a message is issued. For example:

```
enum color { red,
           blue = red+100,      /* ok */
           green = red*0x10     /* 835 */
           };
```

The assignment operators that have an arithmetic or bitwise component, such as `|=`, are also examined. The message given is equivalent to that given with the same operator without the assignment component.

837 switch condition is a constant expression
info
The condition of a `switch` statement is a constant expression as in:

```
switch(5) {
    ...
}
```

While legal, this is suspect since the point of a `switch` statement is usually to specify different actions depending on the value of a variable.

838 previous value assigned to *symbol* not used
info
An assignment statement was encountered that apparently obliterated a previously assigned value that had never had the opportunity of being used. For example, consider the following code fragment:

```
y = 1;
if( n > 0 ) y = 2;
y = 4;               // Informational 838 ...
```

Here we can report that the assignment of `4` to `y` obliterates previously assigned values that were not used. We, of course, cannot report anything unusual about the assignment of 2. This will assign over a prior value of `1` that so far had not been used but the existence of an alternative path means that the value of `1` can still be employed later in the code and is accepted for the time being as reasonable. It is only the final assignment that raises alarm bells. See also Warning message 438.
Supports MISRA C++ Rule 0-1-6 (Req)
Supports MISRA C++ Rule 0-1-9 (Req)

839 storage class of symbol *symbol* assumed static
info
A declaration for a symbol that was previously declared `static` in the same module was found without the '`static`' specifier. For example:

```
static void f();
extern void f();     // Info 839
void f() {}          // Info 839
```

By the rules of the language '`static`' wins and the symbol is assumed to have internal linkage. This could be the definition of a previously declared `static` function (as in line 3 of the above example) in which case, by adding the `static` specifier, you will inhibit this message. This could also be a redeclaration of either a function or a variable (as in line 2 of the above example) in which case the redeclaration is redundant.
Supports MISRA C 2004 Rule 8.11 (Req)

Supports MISRA C 2012 Rule 8.8 (Req)
Supports MISRA C++ Rule 3-3-2 (Req)

840 info NUL character in string literal

A nul character was found in a string literal. This is legal but suspicious and may have been accidental. This is because a nul character is automatically placed at the end of a string literal and because conventional usage and most of the standard library's string functions ignore information past the first nul character.

843 info static storage duration variable *symbol* could be made const

A variable of static storage duration is initialized but never modified thereafter. Was this an oversight? If the intent of the programmer is to not modify the variable, it could and should be declared as `const` [7, Item 3]. See also message 844.

844 info static storage duration variable *symbol* could be made pointer to const

The data pointed to by a pointer of static storage duration is never changed (at least not through that pointer). It therefore would be better if the variable were typed pointer to const [7, Item 3]. See also message 843.

Supports MISRA C 2012 Rule 8.13 (Adv)

845 info the *left/right* operand to *operator* always evaluates to 0

An operand that can be deduced to always be `0` has been presented to an arithmetic operator in a context that arouses suspicion. The name of the operator is provided in the message as well as the side of the operator (`left` or `right`) that had the unusual value. For example:

```
n = 0;
k = m & n;
```

will produce a message that the right hand operand of operator '`&`' is certain to be zero.

The operands examined are the right hand sides of operators

```
+ - | ||
```

the left hand sides of operators

```
/ %
```

and both sides of operators

```
* & << >> &&
```

The reason that the left hand side of operator `+` (and friends) is not examined for zero is that zero is the identity operation for those operators and hence is often used as an initializing value. For example:

```
sum = 0;
for( ... )
    sum = sum + what_ever;   // OK, no message
```

The message is not issued for arithmetic constant zeros. Message 835 is issued in that instance.

The message is also suspended when the expression that evaluates to zero contains side-effects. For example:

```
i = 0;
*(buf + i++) = 'A';     /* Okay */
```

Supports MISRA C 2004 Rule 13.7 (Req)

846 **signedness of bitfield *symbol* of type *type* is implementation-defined**
info A bit-field was detected having the form:

```
int a:5;
```

Most bit fields are more useful when they are `unsigned`. If you want to have a `signed` bit field you must explicitly indicate this as follows:

```
signed int a:5;
```

The same also holds for `typedef`'s. For example,

```
typedef int INT;
typedef signed int SINT;
struct  {
        INT a:16;    // Info 846
        SINT b:16;   // OK
        }:
```

It is very unusual in C or C++ to distinguish between `signed int` and just plain `int`. This is one of those rare cases.

849 **enumerator *symbol* reuses the constant value '*integer*' previously used by enumerator *symbol***
info Two enumerators have the same value. For example:

```
enum colors { red, blue, green = 1 };
```

will elicit this informational message. This is not necessarily an error and you may want to suppress this message for selected enumerators.

850 **for statement index variable *symbol* modified in body**
info A `for` loop with an identifiable loop index variable was programmed in such a way that the loop body also modifies the index variable. For example:

```
void foo(int *a) {
    for (int i = 0; i < 100; i++) {
        a[i++] = 0;
    }
}
```

In general it is better to restrict modifications of `for` loop index variables to the `for` clause if at all possible. If this is not possible, you can prefix the `for` loop with an appropriate lint comment such as:

```
/*lint -e{850} i is modified in the body of the for loop */
```

Supports MISRA C 2004 Rule 13.6 (Req)
Supports MISRA C++ Rule 6-5-3 (Req)

853 **entering nested comment**
info A '`/*`' sequence was encountered inside of a C-style comment while the `fnc` (nested comments) flag was ON. Since nested comments have been enabled, the next '`*/`' sequence that is encountered will only terminate the nested comment, not the containing comment. The purpose of this message is to alert you of this fact.

854 trigraph sequence converted to '*string*' character
info This message is issued when trigraphs are enabled and a trigraph sequence is replaced.
Supports MISRA C 2012 Rule 8.13 (Adv)

855 positional arguments are a non-ISO extension
info Positional arguments are a POSIX extension to C and will not behave as expected on systems that do not support this extension. PC-lint Plus understands and will diagnose misuse specific to positional arguments via messages 493, 494, 2401, and 2404.

856 flag '*string*' is ignored when flag '*string*' is present
info Within a format string for a `printf` or `scanf` like function, a combination of flags was used in which one of the flags has no effect in the presence of the other. For example:

```
extern int i;
printf("%-0d", i);
```

Will elicit the message:

```
flag '0' is ignored when flag '-' is present
```

because a left-justified field (requested via the '-' flag), cannot be padded with zeroes (requested via the '0' flag). Such combinations do not result in undefined behavior but likely represent a programming error.

857 argument 1 of type *type* is not compatible with argument 2 of type *type* in call to function *symbol*
info The first two arguments in a call to `memcmp`, `memmove`, or `memcpy` are not compatible. Using these functions to compare or copy data between different types may have unexpected results.
Supports MISRA C 2012 AMD1 Rule 21.15 (Req)

865 *detail*
info Message 865 is issued as a result of the `-message` option. For example:

```
#ifndef N
//lint -message(Please supply a definition for N)
#endif
```

will issue the message only if `N` is undefined. See option `-message`.

866 unusual argument to sizeof
info An expression used as an argument to `sizeof()` counts as "unusual" if it is not a constant, a symbol, a function call, a member access, a subscript operation (with indices of zero or one), or a dereference of the result of a symbol, scoped symbol, array subscript operation, or function call. Also, since unary '+' could legitimately be used to determine the size of a promoted expression, it does not fall under the category of "unusual". Example:

```
char A[10];
unsigned end = sizeof(A - 1);            // 866: was 'sizeof(A) - 1' intended?
size_of_promoted_char = sizeof(+A[0]);  // OK, + makes a difference
size_t s1 = sizeof( end+1 );             // 866: use +end to get promoted type
size_t s2 = sizeof( +(end+1) );          // OK, we won't complain
struct B *p;                             // B is some POD.
B b1;
```

```
memcpy( p, &b1, sizeof(&b1) );          // 866: sizeof(b1)intended?

size_t s3 = sizeof(A[0]);               // OK, get the size of an element.
size_t s4 = sizeof(A[2]);               // 866: Not incorrect, but unusual ...
```

868 degenerate switch encountered
info
A degenerate switch was encountered. This is a braceless `switch`. E.g.:

Now why, you might wonder, would one want such a thing. That would be to create a region of code from which you can breakout at any point. E.g.:

```
REGION {
       alpha();
       if( n < 10 ) break;
       beta();
       if( n < 25 ) break;
       gamma();
       }
```

If `REGION` is a suitably defined macro then each `break` taken will take you to just below the region. In this simple example there is not that much of an advantage. But when if conditions explode in complexity this is a very nice feature to have.

To obtain this effect you can define `REGION` as

```
#define REGION switch(1) case 1:
```

To automatically suppress this message in this case use:

```
-emacro( 868, REGION )
```

870 no '-max_threads=N' option was encountered prior to the first module; only a single thread will be used by default
info
This message is issued at the end of processing if multiple modules were processed but no `-max_threads` option was used to specify how many concurrent linting threads to employ. By default, PC-lint Plus processes all modules using a single thread. If your hardware has multiple cores or processors, you may be able to substantially speed up the processing time by employing multiple threads using the `-max_threads` option. If a single thread is explicitly requested using `-max_threads=1`, this message will be suppressed.

879 semantic monikers are '*string*' and '*string*'
info
This message is emitted for function calls encountered while the `fsf` flag is enabled. It lists the different ways that the function that was called can be specified within a `-sem`, `-printf`, or `-scanf` option. Overloaded functions and function templates can have multiple ways of being specified in these options. For example, given a function with multiple overloads, it is possible to specify a semantic that applies to all overloads or just one, similarly for function templates. See also 9.2.2.4 Overload-Specific Semantics

882 sizeof applied to parameter *symbol* of function *symbol* declared an incomplete array type *type*
warning
The `sizeof` operator was used with a pointer parameter that was declared using array syntax without a size. Using `sizeof` in this way will yield the size of the pointer, not the size of the array. If an array size if provided, message 682 is issued instead.
Supports MISRA C 2012 AMD1 Rule 12.5 (Mand)

890 **see section *detail* "*detail*" in the manual for details**
supplemental
Provides supplemental information about the location in the manual that should be consulted to address an issue.

891 **reference information *text varies***
supplemental
This supplemental message is used to convey additional information for a previous message at a different location. For example, this message may be used to reference an earlier declaration, a conflicting definition, etc.

892 **did you mean to *multiply/divide* by a factor of type '*strong-type*'?**
supplemental
Provides supplemental information about a Strong Type mismatch when it appears that a forgotten conversion factor may be responsible for a dimensional type difference.

893 **expanded from macro**
supplemental
This supplemental message is given when a message is issued with a location that was the result of a macro expansion. It specifies the macro from which the expansion occurred.

894 **during specific walk *detail***
supplemental
This supplemental message is issued when a value tracking message is issued during a specific walk and provides additional information about the walk. The location of the call, name of the called function, and the arguments passed will be displayed. This information is rendered as described in section 8.5.

896 **semantic expression expands to '*string*'**
supplemental
This supplemental message is issued when there is an error processing a semantic that contains a macro expansion. It provides information about the macro that was expanded.

897 **in instantiation of *string symbol* triggered here**
supplemental
This supplemental message is issued when a message is given within a template instantiation. It provides details of the relevant instantiation.

900 **execution completed producing *integer* primary and *integer* supplemental messages (*integer*
note
total) after processing *integer* module(s)**
This message exists to provide some way of ensuring that an output message is always produced, even if there are no other messages. This is required for some windowing systems and can be useful to distinguish successful completion from premature termination. The message can also be useful for ensuring that all emitted messages have been accounted for. This message can be enable with `+e900`.

901 **variable *symbol* of type *type* not initialized by definition**
note
The definition of the mentioned variable contained no initializer. While the use of an uninitialized variable is caught by warning 530, some style guidelines insist that variables should be initialized at definition. For example, see [8, Rule 19], .

902 **non-static function *symbol* declared outside of a header**
note
A function declaration was found inside a source module and not in a header file. One common programming

practice is to place all function declarations in headers.

904 return statement before end of function *symbol*
note A return statement was found before the end of a function definition. Many programming standards require that functions contain a single exit point located at the end of the function. This can enhance readability and may make subsequent modification less error prone.
Supports MISRA C 2004 Rule 14.7 (Req)
Supports MISRA C 2012 Rule 15.5 (Adv)
Supports MISRA C++ Rule 6-6-5 (Req)

905 non-literal format specifier used (with arguments)
note A `printf`/`scanf` style function received a non-literal format specifier but, unlike the case covered by Warning 592 the function also received additional arguments. E.g.

```
char *fmt;
int a, b;
...
printf( fmt, a, b );
```

Variable formats represent a very powerful feature of C/C++ but they need to be used judiciously. Unlike the case covered by Warning 592, this case cannot be easily rewritten with an explicit visible format. But this Elective Note can be used to examine code with non-literal formats to make sure that no errors are present and that the formats themselves are properly constructed and contain no user-provided data. See Warning 592.

907 implicit conversion to 'void *' from type *type*
note A pointer whose type is not `void*` is being assigned to a variable (or passed to a parameter) whose type is `void*`. This is permitted in both C and C++. But the practice is potentially dangerous and this Elective Note allows one to see where this takes place. See also Note 908.

908 implicit conversion from 'void *' to type *type*
note A pointer whose type is void* is being assigned to a variable (or passed to a parameter) whose type is not void*. This conversion is not permitted in C++ (where Error message 64 is given in lieu of this message). But the conversion is permitted in C. Like the implicit conversion described by Message 907, the practice is potentially dangerous and this Elective Note allows one to see where this takes place.

909 implicit boolean conversion from *type*
note A non-bool was tested as a Boolean. For example, in the following function:

```
int f(int n) {
     if (n) return n;
     else return 0;
}
```

the programmer tests 'n' directly rather than using an explicit Boolean expression such as '`n != 0`'. Some shops prefer the explicit test.
Supports MISRA C++ Rule 5-0-13 (Req)
Supports MISRA C++ Rule 5-0-14 (Req)

910 implicit conversion of null pointer constant to pointer
note A pointer was assigned (or initialized) with a 0. Some programmers prefer other conventions such as `NULL` or

`nil`. This message will help such programmers root out cavalier uses of 0. This is relatively easy in C since you can define NULL as follows:

```
#define NULL (void *)0
```

However, in C++, a `void*` cannot be assigned to other pointers without a cast. Instead, assuming that `NULL` is defined to be `0`, use the option:

```
--emacro((910),NULL)
```

This will inhibit message 910 in expressions using `NULL`.

This method will also work in C. Both methods assume that you expressly turn on this message with a `+e910` or equivalent.
Supports MISRA C 2012 Rule 11.9 (Req)
Supports MISRA C++ Rule 4-10-2 (Req)

911 **implicit promotion from *type* to *type***
note Notes whenever a sub-integer expression such as a `char`, `short`, `enum`, or bit-field is promoted to `int` for the purpose of participating in some arithmetic operation or function call.

912 **implicit promotion of binary operand from *type* to *type***
note Notes whenever a binary operation (other than assignment) requires a type balancing. A smaller range type is promoted to a larger range type. For example: `3 + 5.5` will trigger such a message because `int` is converted to `double`.

915 **implicit arithmetic conversion (*context*) from *type* to *type***
note Notes whenever an assignment, initialization or `return` implies an arithmetic conversion (*context* specifies which).

916 **implicit pointer assignment conversion (*context*) from *type* to *type***
note Notes whenever an assignment, initialization or `return` implies an implicit pointer conversion (*context* specifies which).

917 **implicit conversion from *type* to *type* due to function prototype**
note Notes whenever an implicit arithmetic conversion takes place as the result of a prototype. For example:

```
double sqrt(double);
... sqrt(3); ...
```

will elicit this message because `3` is quietly converted to `double`.

919 **implicit conversion (*context*) from lower precision type *type* to higher precision type *type***
note A lower precision quantity was assigned to a higher precision variable as when an `int` is assigned to a `double`.

920 **explicit cast from *type* to *type***
note A cast is being made to `void`.
Supports MISRA C++ Rule 5-2-7 (Req)

921 explicit cast from *type* to *type*
note A cast is being made from one integral type to another.

922 explicit cast from *type* to *type*
note A cast is being made to or from one of the floating types (`float`, `double`, `long double`).

923 explicit cast from *type* to *type*
note A cast is being made from a pointer to a non-pointer or from a non-pointer to a pointer.
Supports MISRA C 2004 Rule 11.3 (Adv)
Supports MISRA C 2012 Rule 11.6 (Req)
Supports MISRA C++ Rule 5-2-7 (Req)
Supports MISRA C++ Rule 5-2-8 (Req)

925 explicit cast from *type* to *type*
note A cast is being made between pointers wherein the source or destination type is a pointer to `void`.
Supports MISRA C++ Rule 5-2-8 (Req)

926 explicit cast from *type* to *type*
note A cast is being made between pointers to (possibly `signed` or `unsigned`) `char`.
Supports MISRA C++ Rule 5-2-7 (Req)

927 explicit cast from *type* to *type*
note A cast is being made from pointer to (possibly `signed` or `unsigned`) `char` to a pointer to another type.
Supports MISRA C++ Rule 5-2-7 (Req)

928 explicit cast from *type* to *type*
note A cast is being made to pointer to (possibly `signed` or `unsigned`) `char` from a pointer to another type.
Supports MISRA C++ Rule 5-2-7 (Req)

929 explicit cast from *type* to *type*
note A cast is being made between pointers that does not fall under the purview of 920, 922, 923, 925, 927, 928.
Supports MISRA C++ Rule 5-2-7 (Req)

930 explicit cast from *type* to *type*
note A cast is being made to or from an enumeration type.
Supports MISRA C++ Rule 5-2-7 (Req)
Supports MISRA C++ Rule 5-2-8 (Req)

931 both sides have side effects
note Indicates when both sides of an expression have side-effects. An example is `n++ + f()`. This is normally benign. The really troublesome cases such as `n++ + n` are caught via Warning 564. Also, if the function `f()` modifies `n` then this will be reported in pass 2 as Warning `591`.
Supports MISRA C 2012 Rule 13.2 (Req)

935 data member *symbol* declared as type *type*
note This Note helps to locate non-portable data items within `struct`'s. If instead of containing `int`'s and `unsigned int`'s, a `struct` were to contain `short`'s and `long`'s then the data would be more portable across machines and memory models. Note that bit fields and `union`'s do not get complaints.

936 old-style function definition for function *symbol*
note An "old-style" function definition is one in which the types are not included between parentheses. Only names are provided between parentheses with the type information following the right parenthesis. This is the only style allowed by K&R.
Supports MISRA C 2012 Rule 8.2 (Req)

937 old-style function declaration for function *symbol*
note An "old-style" function declaration is one without type information for its arguments.
Supports MISRA C 2004 Rule 8.1 (Req)
Supports MISRA C 2004 Rule 16.5 (Req)
Supports MISRA C 2012 Rule 8.2 (Req)

940 omitted braces within initializer
note An initializer for a subaggregate does not have braces. For example:

```
int a[2][2] = { 1, 2, 3, 4 };
```

This is legal C but may violate local programming standards. The worst violations are covered by Warning 651.
Supports MISRA C 2004 Rule 9.2 (Req)
Supports MISRA C++ Rule 8-5-2 (Req)

941 result 0 due to operand(s) equaling 0 in operation '*operator*'
note The result of a constant evaluation is `0` owing to one of the operands of a binary operation being `0`. This is less severe than Info 778 wherein neither operand is `0`. For example, expression `(2&1)` yields a 778 whereas expression `(2&0)` yields a `941`.

942 possibly truncated *string* promoted to *type*
note An integral expression (signed or unsigned) involving addition or subtraction is converted to a floating point number. If an overflow occurred, information would be lost. See also messages 647, 776 and 790.

943 too few initializers for aggregate *symbol* of type *type*
note The initializer `{0}` was used to initialize an aggregate of more than one item. Since this is a very common thing to do, it is given a separate message number, which is normally suppressed. See 785 for more flagrant abuses.

944 *left/right* operand to '*operator*' always evaluates to '*true/false*'
note The indicated operator (which is either `&&`, `||`, or `!`) has an argument that appears to always evaluate to either `true` or `false`. Information is gleaned from a variety of sources including prior assignment statements and initializers. Compare this with message 506, which is based on testing constants or combinations of constants.
Supports MISRA C++ Rule 0-1-1 (Req)

Supports MISRA C++ Rule 0-1-2 (Req)
Supports MISRA C++ Rule 0-1-9 (Req)

945 variable of undefined structure type declared to be extern
note Some compilers refuse to process declarations of the form:

```
extern struct X s;
```

where `struct X` is not yet defined. This note can alert a programmer porting to such platforms.

946 relational operator *string* applied to pointers
note A relational operator (one of >, >=, <, <=) or the subtract operator has been applied to a pair of pointers. The reason this is of note is that when large model pointers are compared (in one of the four ways above) or subtracted, only the offset portion of the pointers is subject to the arithmetic. It is presumed that the segment portion is the same. If this presumption is not accurate then disaster looms. By enabling this message you can focus in on the potential trouble spots.
Supports MISRA C 2012 Rule 18.3 (Req)
Supports MISRA C++ Rule 5-0-18 (Req)

947 pointer subtraction
note An expression of the form `p - q` was found where both `p` and `q` are pointers. This is of special importance in cases where the maximum pointer can overflow the type that holds pointer differences. For example, suppose that the maximum pointer is 3 Gigabytes -1, and that pointer differences are represented by a `long`, where the maximum `long` is 2 Gigabytes -1. Note that both of these quantities fit within a 32 bit word. Then subtracting a small pointer from a very large pointer will produce an apparent negative value in the `long` representing the pointer difference. Conversely, subtracting a very large pointer from a small pointer can produce a positive quantity.

The alert reader will note that a potential problem exists whenever the size of the type of a pointer difference equals the size of a pointer. But the problem doesn't usually manifest itself since the highest pointer values are usually less than what a pointer could theoretically hold. For this reason, the message cannot be given automatically based on scalar types and hence has been made an Elective Note.

Compare this Note with that of 946, which was designed for a slightly different pointer difference problem.
Supports MISRA C 2012 Rule 18.2 (Req)

948 operator *operator* always evaluates to *true/false*
note The operator named in the message is one of four relational operators or two equality operators in the list:

```
>   >=   <   <=
==  !=
```

The arguments are such that it appears that the operator always evaluates to either `true` or to `false` (as indicated in the message). This is similar to message 944. Indeed there is some overlap with that message. Message 944 is issued in the context where a Boolean is expected (such as the left hand side of a `?` operator) but may not involve a relational operator. Message 948 is issued in the case of a relational (or equality) operator but not necessarily in a situation that requires a Boolean.
Supports MISRA C++ Rule 0-1-9 (Req)

952 parameter *symbol* of function *symbol* could be const
note A parameter is not modified by a function. For example:

```
int f( char *p, int n ) { return *p = n; }
```

can be redeclared as:

```
int f( char * const p, const int n ) { return *p = n; }
```

There are few advantages to declaring an unchanging parameter a `const`. It signals to the person reading the code that a parameter is unchanging, but, in the estimate of most, reduces legibility. For this reason the message has been given an Elective Note status.

However, there is a style of programming that encourages declaring parameters const. For the above example, this style would declare `f` as

```
int f( char *p, int n);
```

and would use the `const` qualifier only in the definition. Note that the two forms are compatible according to the standard. The declaration is considered the interface specification where `const` does not matter. The `const` does matter in the definition of the function, which is considered the implementation. Message 952 could be used to support this style.

Marking a parameter as `const` does not affect the type of argument that can be passed to the parameter. In particular, it does not mean that only `const` arguments may be passed. This is in contrast to declaring a parameter as pointer to `const` or reference to `const`. For these situations, Informational messages are issued (818 and 1764 respectively) and these do affect the kinds of arguments that may be passed. See also messages 953 and 954.
Supports MISRA C++ Rule 7-1-1 (Req)

953 local variable *symbol* could be const
note
A local auto variable was initialized and referenced but never modified. Such a variable could be declared `const`. One advantage in making such a declaration is that it can furnish a clue to the program reader that the variable is unchanging. Other situations in which a `const` can be added to a declaration are covered in messages 818, 843, 844, 952, 954 and 1764.

954 local variable *symbol* could be pointer to const
note
The data pointed to by a pointer is never changed (at least not through that pointer). It may therefore be better, or at least more descriptive, if the variable were typed pointer to `const`. For example:

```
{
char *p = "abc";
for( ; *p; p++ ) print(*p);
}
```

can be redeclared as:

```
{
const char *p = "abc";
for( ; *p; p++ ) print(*p);
}
```

It is interesting to contrast this situation with that of pointer parameters. The latter is given Informational status (818) because it has an effect of enhancing the set of pointers that can be passed into a function. Other situations in which a `const` can be added to a declaration are covered in messages 952, 953 and 1764.
Supports MISRA C 2012 Rule 8.13 (Adv)

955 parameter *integer* (*type*) of forward declaration of *symbol* lacks a name
note
In a function declaration a parameter name is missing. For example:

```
void f(int);
```

will raise this message. This is perfectly legal but misses an opportunity to instruct the user of a library routine on the nature of the parameter. For example:

```
void f(int count);
```

would presumably be more meaningful. [9, Rule 34]

This message is not given for function definitions, only function declarations.
Supports MISRA C 2004 Rule 16.3 (Req)
Supports MISRA C 2012 Rule 8.2 (Req)

956 ***string* variable *symbol* of type *type* is neither const nor atomic**
note This check has been advocated by programmers whose applications are multi-threaded. Software that contains modifiable data of static duration is often non-reentrant. That is, two or more threads cannot run the code concurrently. By 'static duration' we mean variables declared static or variables declared external to any function. For example:

```
int count = 0;
void bump() { count++; }
int get_count() { return count; }
```

If the purpose is to obtain a count of all the `bump()`'s by a given thread then this program clearly will not do since the global variable `count` sums up the `bump()`'s from all the threads. Moreover, if the purpose of the code is to obtain a count of all `bump()`'s by all threads, it still may contain a subtle error (depending on the compiler and the machine). If it is possible to interrupt a thread between the access of `count` and the subsequent store, then two threads that are `bump()`'ing at the same time, may register an increase in the `count` by just one.

Please note that not all code is intended to be re-entrant. In fact most programs are not designed that way and so this Elective Note need not be enabled for the majority of programs. If the program is intended to be re-entrant, all uses of non-const static variables should be examined carefully for non-reentrant properties.

957 **function *symbol* defined without a prototype in scope**
note A function was defined without a prototype in scope. It is usually good practice to declare prototypes for all functions in header files and have those header files checked against the definitions of the function to assure that they match.

If you are linting all the files of your project together such cross checking will be done in the natural course of things. For this reason this message has been given a relatively low urgency of Elective Note.
Supports MISRA C 2004 Rule 8.1 (Req)
Supports MISRA C 2012 Rule 8.4 (Req)

958 **padding of *integer* bytes needed to align *string* on a *integer* byte boundary**
note This message is given whenever padding is necessary within a `struct` to achieve a required member alignment. *symbol* designates that which is being aligned. Consider:

```
struct A { char c; int n; };
```

Assuming that `int` must be aligned on a 4-byte boundary and assuming the size of a `char` to be 1, then this message will be issued indicating that there will be a padding of 3 bytes to align the number.

The alignment requirements vary with the compiler, the machine and, sometimes, compiler options. When separately compiled programs need to share data at the binary level it helps to remove any artificially created padding from any of the structures that may be shared.

959 note **nominal structure size of '*integer*' bytes is not a whole multiple of its alignment of '*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.

963 note **qualifier '*string*' follows a type; use -fqb to reverse the test**

The declarations in the following example are equivalent:

```
//lint +e963 report on qualifier-type inversion
extern const char *p;
extern char const *p;   // Note 963
```

The qualifiers '`const`' and '`volatile`' may appear either before or after or even between other declaration specifiers. Many programmers prefer a consistent scheme such as always placing the qualifier before the type. If you enable `963` (using `+e963`) this is what you will get by default. The message will contain the word '`follows`' rather than the word '`precedes`'.

There is a diametrically opposite convention, viz. that of placing the qualifier after the type. As the message itself reminds the user, you will obtain the reverse test if you turn off the `fqb` (place qualifiers before types) flag. Thus

```
//lint -fqb turn off the Qualifiers Before types flag
//lint +e963 report on type-qualifier inversion
extern const char *p; // Note 963
extern char const *p;
```

Note that the use of this flag will cause '`follows`' in the message to be replaced by '`precedes`' and the alternative option mentioned within the '`use`' clause is changed to its opposite orientation.

Dan Saks [2] and Vandevoorde and Josuttis, [3], section 1.4, provide convincing evidence that this alternative convention is indeed the better one.

967 note **header file '*string*' does not have a standard include guard**

You may protect against the repeated inclusion of headers by means of a standard include guard having the following form:

```
#ifndef Name
#define Name
...
#endif
```

or

```
#if !defined(Name)
#define Name
...
```

```
#endif
```

The header file cited in the message does not have such a guard. It is standard practice in many organizations to always place include guards within every header.

See message 451 for more information about header include guards. This message is not issued for headers that employ `#pragma` once.
Supports MISRA C++ Rule 16-2-3 (Req)

970 use of modifier or type '*name*' outside of a typedef

note Some standards require the use of type names (defined in `typedef`'s) in preference to raw names used within the text of the program. For example they may want you to use `INT32` rather than `int` where `INT32` is defined as:

```
typedef int INT32;
```

This message is normally issued for the standard intrinsic types: `bool`, `char`, `wchar_t`, `int`, `float`, `double`, and for modifiers `unsigned`, `signed`, `short` and `long`. You may enable this message and then suppress the message for individual types to obtain special effects. For example, the following will enable the message for all but `bool`.

```
+e970 -estring(970,bool)
```

Supports MISRA C 2004 Rule 6.3 (Adv)
Supports MISRA C 2012 Directive 4.6 (Adv)
Supports MISRA C++ Rule 3-9-2 (Adv)

971 use of plain char

note The '`char`' type was specified without an explicit modifier to indicate whether the `char` was `signed` or `unsigned`. The plain `char` type can be regarded by the compiler as identifying a `signed` or an `unsigned` quantity, whichever is more efficient to implement. Because of this ambiguity, some standards do not like the use of `char` without an explicit modifier to indicate its signedness.

972 unusual character '*string*' in '*kind*' literal

note An unusual character was found in a character or string literal. It is identified in the message by its hexadecimal encoding. Characters are considered to be unusual if they are outside the standard C and C++ source character set. For example:

```
char ch = '\`'; // Unusual character '\\x60'
```

The backtick character being assigned above is considered unusual. To suppress this message for this character use the option.

```
-estring( 972, "\\{x60}" )
```

974 worst case stack usage: *detail*

note This message, issued at global wrap-up, will report on the function that requires the most stack. The stack required consists of the amount of auto storage the function requires plus the amounts required in any chain of functions called. The worst case chain is always reported.

To obtain a report of all the functions, use the `+stack` option.

Reasonable allowances are made for function call overhead and the stack requirements of external functions. These assumptions can be controlled via the `+stack` option.

If recursion is detected it will be reported here, as this is considered worse than any finite case. The next worst case is that the stack can't be determined because a function makes a call through a function pointer. The function is said to be non-deterministic. If neither of these conditions prevail, the function that heads the worst case chain of calls will be reported upon.

The message will normally provide you with the name of a called function. If the function is recursive this will provide you with the first call of a recursive loop. To determine the full loop, you will need a full stack report as obtained with the `+stack` option. You need a suboption of the form `&file=`*file* to specify a file that will contain a record for each function for which a definition was found. You will be able to follow the chain of calls to determine the recursive path.

If you can assure yourself through code analysis that there is an upper bound to the amount of stack utilized by some recursive function, then you can employ the `+stack` option to specify the bound for this function. The function will no longer be considered recursive but rather finite. In this way, possibly through a sequence of options, you can progressively eliminate apparent recursion and in that way arrive at a safe upper bound for stack usage. Similar considerations apply for non-deterministic functions.

975 **unknown pragma '*string*' will be ignored**
note The first identifier after `#pragma` is considered the name of the pragma. If the name is unrecognized then the remainder of the line is ignored. Since the purpose of `#pragma` is to allow for compiler- dependent communication, it is not really expected that all pragmas will be understood by all third-party processors of the code. Thus, this message does not necessarily indicate that there is anything wrong and could easily be suppressed entirely.

Moreover, if the pragma occurs in a library header, this message would not normally be issued because the option `-wlib(1)` would be in effect (this option is present in all of our compiler options files).

But, if the pragma occurs in user code, then it should be examined to see if there is something there that might interest a lint processor. There are a variety of facilities to deal with pragmas; in particular, they can be mapped into linguistic constructs or lint options or both. See Section 4.11.5 Pragmas.
Supports MISRA C 2004 Rule 3.4 (Req)

977 **non-literal non-boolean used in *type string***
note This message is issued when a non-literal expression of non-boolean type is assigned to a boolean. This can occur through direct assignment, initialization, as an argument in a function call, or a return expression. For example, the below function returns true if there is a remainder when `x` is divided by `y` but the type of the value before conversion is `int`:

```
_Bool foo(int y, int z) {
     return y % z;       // Note 977
}
```

The message won't be issued for:

```
_Bool foo(int y, int z) {
     return y % z != 0;  // Okay, != implies boolean context
}
```

A cast can also be used to suppress this message.

978 **the name '*name*' matches a pattern reserved to the compiler *string***
note The C Standard specifies a variety of naming patterns reserved for future use. For example, names starting with *is*, *to*, or *str* followed by a lowercase letter are reserved to the implementation. This message reports on

symbols declared with names that match one of these patterns. The name of the offending symbol is provided along with the reserved pattern that the name matches. For example:

```
int strmin;
```

will elicit:

```
note 978: the name 'strmin' matches a pattern reserved to the compiler because
    it begins with 'str' and a following lowercase letter
int strmin;
    ^
```

979 function *symbol* could be marked with a 'pure' semantic
note The specified function was analyzed and determined to be eligible for the `pure` semantic but no `-sem` option was used to specify that this function was `pure`. Since functions are considered to be impure by default when the definition is not visible from the current module, specifying this function as `pure` could improve analysis related to side-effects and purity.

980 macro name '*name*' matches a pattern reserved to the compiler *string*
note The C Standard specifies a variety of macro naming patterns reserved by the implementation. These patterns include a name starting with '`E`' followed by a digit or upper case letter, names starting with '`SIG`' or '`SIG_`' followed by an uppercase letter, and macros that begin with '`PRI`' or '`SCN`' followed by either '`X`' or a lowercase letter. The message includes both the name of the offending macro and the reserved pattern that it matches. For example:

```
#define LC_END
```

will be greeted with:

```
note 980: macro name 'LC_END' matches a pattern reserved to the compiler
    because it begins with 'LC_' and a following uppercase letter
#define LC_END
        ^
```

Note that some patterns are reserved only in certain versions of C and will be diagnosed only when the language mode specified corresponds to the version in which the pattern is applicable. For example, names starting with `INT` and ending in `_C` are diagnosed only in C99 and later.
Supports MISRA C 2004 Rule 20.1 (Req)

981 cast of expression of type *type* to same type is redundant
note A cast is being performed on an expression that is already of the type being cast to making the cast redundant. This message is not issued for casting enumerations to their underlying type.

983 behavior of dash in scan list is implementation defined
note A dash (`-`) was encountered within the scan list in a `%[` conversion specifier in the call to a `scanf`-like function. Furthermore the dash was not the first character (or the second character where the first character is `^`) or the last character, e.g. `%[A-Z]`. The behavior of a dash in this position is implementation defined, some implementations interpret this as a range of characters to include in the scan set (e.g. all characters from `A` to `Z`) while others treat it literally.

986 target type *type* of type alias *symbol* is deprecated
note This message is issued when a type that has been deprecated using the `-deprecate` option with a category of *type* is used as a target in a `typedef` definition. This is to provide notification that the underlying deprecated

type may be used through a `typedef` later, which will not be diagnosed. To diagnose uses of a type through a `typedef`, the `basetype` deprecation category can be used. See the `-deprecate` option for more information.

987 **constructor parameter *symbol* shadows the field *symbol* of *symbol***
note This message is a weaker form of message `578` for cases where a field is shadowed by a constructor parameter.

999 **defaulting to *string* concurrent threads**
note The `-max_threads=`*n* option can be used to specify the number of concurrent linting threads for parallel analysis. If *n* is specified as 0, PC-lint Plus attempts to query the hardware to determine the optimal value for *n*. This message serves to inform the programmer of the value that was selected.

15.3 Messages 1000-1999

1001 **scope *string* must be a class*string***
error In an expression of the form `X::Y, X` must be a class name. [10, Section 10.4]

1002 **'this' must be used in a non-static class member function**
error The keyword `this` refers to the class being passed implicitly to a member function. It is invalid outside a class member function. [10, Section 5.1]

1003 **'this' cannot be used in a static member function declaration**
error A `static` member function receives no `this` pointer. [10, Section 9.5]

1004 **right hand operand to '*string*' has non-pointer-to-member type *type***
error The `.*` and `->*` operators require pointer to members on the right hand side. [10, Section 5.5]

1005 **destructor declaration requires a class**
error While expecting a declaration a '~' character was encountered. This was presumed to be the start of a destructor. However no class was specified. [10, Section 12.4]

1008 **initializer on function does not look like a pure-specifier**
error Some nonstandard extensions to C++ allow integers to follow '=' for declarations of member functions. If you are using such extensions, simply suppress this message. If only library headers are using this extension, use `-elib(1008)`. [10, Section 10.3]

1012 **return type not allowed for conversion function**
error The return type of a function introduced with '`operator` *type*' is *type* and may not be preceded with the same or any other type. [10, Section 12.3.2]

1013 **symbol *symbol* is not a member of *string***
error The second operand of a scope operator or a '`.`' or '`->`' operator is not a member of the `class` (`struct` or `union`) expressed or implied by the left hand operand. [10, Section 3.2]

1020 **template specialization declared without a 'template<>' prefix**
error A class template specialization is generally preceded by a 'template' clause as in:

```
template< class T > class A { };   // a template
template<> class A<int> { };       // a specialization
```

If the 'template<>' is omitted you will get this message but it will still be interpreted as a specialization. Before the standardization of template syntax was completed, a template specialization did not require this clause and its absence is still permitted by some compilers.

1022 **function *symbol* must be a non-static member function**
error There are four operators not to be defined except as class members. These are:

```
= () [] ->
```

The parameter *symbol* indicates which it is. [10, Section 13.4.3 and 13.4.6]

1023 **call to *symbol* is ambiguous**
error A call to an overloaded function or operator is ambiguous. The candidates of choice are provided in the message. [10, Section 13.2]

1024 **no matching function for call to *symbol***
error A call to an overloaded function could not be resolved successfully because no function is declared with the same number of arguments as in the call. [10, Section 13.2]

1027 **missing default argument for parameter *symbol* of function *symbol***
error Default arguments need to be consecutive. For example

```
void f(int i=0, int j, int k=0);
```

is illegal. [10, Section 8.2.6]

1029 **default argument repeated for parameter *symbol* in function *symbol***
error A default value for a given argument for a given function should be given only once. [10, Section 8.2.6]

1031 **local variable *symbol* used in default argument expression**
error Default values for arguments may not use local variables. [10, Section 8.2.6]

1032 **non-static member function *symbol* cannot be called without object**
error There was an attempt to call a non-static member function without specifying or implying an object that could serve as the basis for the `this` pointer. If the member name is known at compile time it will be printed with the message. [10, Section 5.24]

1033 **static member functions cannot be virtual**
error You may not declare a static member function `virtual`. [10, Section 10.2]

1034 **'static' can only be specified inside the class definition**
error This can come as a surprise to the novice C++ programmer. The word '`static`' within a class definition is used to describe a member that is alone and apart from any one object of a class. But such a member has program scope not file scope. The word '`static`' outside a class definition implies file scope not program scope. [10, Section 9.5]

1036 **call to constructor of *symbol* is ambiguous**
error There is more than one constructor that can be used to make a desired conversion. [10, Section 12.3.2]

1037 **conversion *between types* is ambiguous**
error There is more than one conversion function (of the form `operator` *type* `()`) that will perform a desired conversion. [10, Section 12.3.2]

1038 **type *type* not found, *type* assumed**
error We have found what appears to be a reference to a type but no such type is in scope. We have, however, been able to locate a type buried within another class. Is this what the user intended? If this is what is intended, use full scoping. If your compiler doesn't support the scoping, suppress with `-esym`. [10, Section 3.2]

1040 **non-friend class member *symbol* cannot have a qualified name**
error A declaration of the symbol `X::Y` appears within a class definition (other than for class `X`). It is not a `friend` declaration. Therefore it is in error.

1042 **at least one class-like parameter is required for overloaded *string***
error In defining (or declaring) an operator you must have at least one class as an operand. [10, Section 13.4]

1043 **cannot delete expression of type *type***
error An expression being `delete`'d is a non-pointer, non-array. You may only `delete` what was created with an invocation of `new`. [10, Section 5.3.4]

1046 **member *symbol*, referenced in a static function, requires an object**
error The *symbol* is a non-static member of a class and hence requires a class instantiation. None is in sight. [11, Section `class.static`]

1049 ***too few/too many* template arguments for *template-kind symbol***
error There are more arguments in the template class-name than there were parameters in the original template declaration. [11, Section `temp.decls`]

1050 **use of class template *symbol* requires template arguments**
error The name of a class template identified by *symbol* was used without specifying a template argument list. [11, Section `temp.decl`]

1051 **redefinition of *symbol* as different kind of symbol**
error Whereas it is possible to overload a function name by giving it two different parameter lists, it is not possible

to overload a name in any other way. In particular, a function name may not also be used as a variable name. [10, Section 9.2]

1055 **use of undeclared identifier *string*; did you mean *symbol*?**
error Whereas in C you may call a function without a prior declaration, in C++ you must supply such a declaration. For C programs you would have received an Informational message (718) in this event. [10, Section 5.2.2]

1057 **member *symbol* cannot be used without an object**
error The indicated member referenced via scope operator cannot be used in the absence of a `this` pointer. [10, Section 5.2.4]

1063 **copy constructor for class *symbol* must pass its first argument by reference**
error A constructor for a class closely resembles a copy constructor. A copy constructor for class `X` is typically declared as:

```
X(const X &)
```

If you leave off the '`&`' then a copy constructor would be needed just to copy the argument into the copy constructor. This is a runaway recursion. [10, Section 12.1]

1069 **member initializer *string* does not name a non-static data member or base class of class *symbol***
error Within a constructor initialization list, a name was found that did not correspond to either a direct base class of the class being defined or a member of the class.

1071 **constructor cannot have a return type**
error Constructors and destructors may not be declared with a return type, not even `void`. See ARM [10, Section 12.1 and 12.4]

1072 **reference variable *string* must be initialized**
error A reference variable must have an initializer at the point of declaration.

1074 **expected a namespace identifier**
error In a declaration of the form:

```
namespace name = scoped-identifier
```

the *scoped-identifier* must identify a namespace.

1075 **ambiguous reference to symbol *symbol***
error Two namespaces contain the same name. A reference to such a name could not be disambiguated. You must fully qualify this name in order to indicate the name intended.

1076 **anonymous union assumed to be 'static'**
error Anonymous unions need to be declared `static`. This is because the names contained within are considered local to the module in which they are declared.

1079 **could not find '>' or ',' to terminate template parameter**
error
The default value for a template parameter appears to be malformed. For example, suppose the user mistakenly substituted a ']' for a '>' producing the following:

```
template <class T = A< int ] >
    class X
    {
    };
```

This will cause PC-lint Plus to process to the end of the file looking (in vain) for the terminating pointy bracket. Not finding it will cause this message to be printed. Fortunately, the message will bear the *location* of the malformed template.

1083 **ambiguous conversion between 2nd and 3rd arguments of conditional operator;** ***reason***
error
If the 2nd operand can be converted to match the type of the 3rd, and the 3rd operand can be converted to match the type of the 2nd, then the conditional expression is considered ill-formed.

1088 **a using declaration requires a qualified-id**
error
This error is issued when a using-declaration references a name without the `::` scope resolution operator; e.g.:

```
class A {
protected:
    int n;
};

class B : public A {
public:
    using n; // Error 1088: should be 'using A::n;'
};
```

See [12, Section 7.3.3 `namespace.udecl`].

1089 **a using declaration must not name a namespace**
error
This error is issued when the rightmost part of the qualified-id in a using-declaration is the name of a namespace. E.g.:

```
namespace N {
    namespace Q {
        void g();
    }
}

void f() {
    using ::N::Q; // Error 1089
    Q::g();
}
```

Instead, use a namespace-alias-definition:

```
namespace N {
    namespace Q {
        void g();
    }
}
```

```
void f() {
    namespace Q = ::N::Q;    // OK
    Q::g();                  // OK, calls ::N::Q::g().
}
```

See [13], Issue 460.

1090 **a using declaration must not name a template-id**
error This error is issued when the rightmost part of the qualified-id in a using-declaration is a template-id. E.g.:

```
template <class T> class A {
protected:
    template <class U> class B {};
};

struct D : public A<int> {
public:
    using A<int>::B<char *>; // Error 1090
};

D::B<char *> bc;
```

Instead, refer to the template name without template arguments:

```
template <class T> class A {
protected:
    template <class U> class B {};
};

struct D : public A<int> {
public:
    using A<int>::B; // OK
};

D::B<char *> bc;     // OK
```

See [12], 7.3.3 `namespace.udecl`.

1091 **using declaration refers into *symbol*, which is not a base class of *symbol***
error This error is issued when the nested-name-specifier of the qualified-id in a using-declaration does not name a base class of the class containing the using-declaration; e.g.:

```
struct N {
    void f();
};

class A {
protected:
    void f();
};

class B : A {
public:
    using N::f;     // Error 1091
};
```

See [13], Issue 400.

1092 **a using declaration that names a class member must be a member declaration**
error
This error is issued when the nested-name-specifier of the qualified-id in a using-declaration names a class but the using-declaration does not appear where class members are declared. E.g.:

```
struct A {
    void f();
};

struct B : A {
    void g() { using A::f; }    // Error 1092
};
```

See [12], 7.3.3 `namespace.udecl`.

1093 ***symbol* is not virtual and cannot be declared pure**
error
A pure specifier (`"= 0"`) should not be placed on a function unless the function had been declared `"virtual"`.

1096 **an initializer for a delegating constructor must appear alone**
error
C++11 requires that if a constructor delegates to another constructor, then the *mem-initializer* (the region between the colon and the function body) must contain only one item, and that item must be a call to another constructor (which is called the "target constructor"). Example

```
struct A {
    int n;
    A(int);
    A(const A &p) : A(p.n) { }   // OK
    A() : n(42), A(32) { }       // Error 1096
};
```

1097 **constructor *symbol* creates a delegation cycle**
error
The specified constructor is a delegating constructor that contains a delegation cycle, either directly by delegating to itself or indirectly by calling another delegating constructor that eventually delegates back to the original constructor. If multiple constructors are in the cycle, the other constructors are provided via subsequent `891` supplemental messages. For example:

```
struct A {
    int n;
    A(int x) : A(x, 0) { }              // Error 1097
    A(int x, int y) : A(x, y, 0) { }
    A(int x, int y, int z) : A(x) { }
};
```

1098 **function template specialization *symbol* does not match any function template**
error
This message is issued for a declaration where the user apparently intended to name a specialization of a function template (e.g., in an explicit specialization, an explicit instantiation or a friend declaration of specialization), but no previously-declared function template is matched. Example:

```
template<class T> void f( const T& ); // #1
```

```
struct A{};
template<> void f( const A& );        // Ok
// (A is the deduced argument to T.)

struct B{};
template<> void f( const B );         // Error 1098.
// (A template argument cannot be deduced for T.)
```

1099 error **function template specialization *symbol* ambiguously refers to more than one function template; explicitly specify template arguments to identify a particular function template**

This message is issued for a declaration where the user apparently intended to name a specialization of a function template (e.g., in an explicit specialization, an explicit instantiation or a friend declaration of specialization), but the specialization matches multiple function templates, and none of the matched templates is more specialized than all of the other matching templates. The candidates (i.e., the matching templates) are provided in the message. Example:

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

template<class T, class U> void f( T*, U   ); // #1
template<class T, class U> void f( T, A<U> ); // #2

struct B{};
template<> void f( B, A<B> );                 // Ok
// #1 does not match but #2 does.

template<> void f( char*, A<int> );           // Error 1099
// Both #1 and #2 match and neither is more specialized than the other.
```

This situation can be avoided in at least a couple of ways. One way is to explicitly specify one or more template arguments. Example

```
// continuing from above...
template<> void f<char*>( char*, A<int> ); // Ok
// #1 does not match but #2 does.
```

Another way is to use SFINAE (Substitution Failure Is Not An Error) tactics in the declaration of one or more function templates, e.g. with `boost::enable_if`.

1101 error **type of variable *symbol* cannot be deduced from its initializer**

Example:

```
int f(void);
int f(char*);
auto n = f; // Error
```

In terms of deduction, this is equivalent to:

```
int f(void);
int f(char *);
template <class T> void g(const T &);

void h(void) {
g(f);  // Error
}
```

Here, '`f`' refers to multiple overloaded functions, so it is an ambiguous reference and `T` cannot be deduced. (Code like this could still be well-formed however, e.g. if `g` is overloaded with a non-template function whose parameter type is 'ptr-to-function returning `int` taking `char*`'.)

1102 error ***string*' type deduced inconsistently: *type* for *symbol* but *type* for *symbol***

When multiple variables are defined in the same declaration, and when that declaration uses the keyword `auto` as the *type-specifier*, the type for which `auto` is a placeholder must be the same for each variable. Example:

```
float g(void);
char* s();
auto a = 42;            // Ok, auto is 'int'
auto b = g();           // Ok, auto is 'float'
auto c = 'q',
     *d = s();          // Ok, auto is 'char' (for both c and d)
auto x = 42, y = g();   // Error 1102 here
```

1103 error **non-integral type *type* is not a valid enum-base**

When an enumeration type is declared with an explicit underlying type, that type must be integral. Example:

```
enum A : bool;                // ok
enum B : short;               // ok
enum C : unsigned long long;  // ok
enum D : float;               // Error 1103
```

1105 error **cannot overload a member function *with a certain ref-qualifier* with a member function *with different ref-qualifiers***

If an explicit ref qualifier ('`&`' or '`&&`') of a nonstatic member function is employed, an explicit ref qualifier needs to be used with every member of the overload set. Thus:

```
class A {
    void f(int)&;
    void f(int);
    void f(double);
    void g(int);
    void g(double);
};
```

1107 error **invalid concatenation of wide string literals of different kinds**

Two string literals that different types are being concatenated. Examples:

```
char *s = u"abc" U"def";
char *q = L"ghi" u"jkl";
```

This message is issued for mixing strings of `char16_t`, `char32_t`, and/or `wchar_t` (as shown). Literal string concatenation of any of these with an ordinary character literal is permitted and will receive Informational 707.

1108 error **use of *unavailable/deleted* function *symbol***

This message is issued when a deleted or otherwise unavailable function is used. For example:

```
void f( int ) = delete;
void f( double );
void g( double d, int n ) {
    f( d ); // Ok
    f( n ); // Error
}
```

1110 error **circular pointer delegation detected: explicit application of *type*::operator-> causes infinite applications of the same operator**

When an overloaded `operator->` is used as in

```
a->b
```

it is effectively expanded to:

```
a.operator->()->b
```

And this expansion repeats until an `operator->` is found that does not yield a class type. But in the process of evaluating this expansion, it might be found that one of the operators returns a class type for which an overloaded `operator->` was already expanded; in that case, Error `1110` is triggered. Example:

```
struct B;
struct A { struct B& operator->(); };
struct B { struct A& operator->(); };
int f( A & p ) { p->g(); }            // Error
```

1111 error **explicit specialization of *type* must appear at namespace scope**

This message is issued at the beginning of each explicit specialization/instantiation that does not appear at namespace scope. Example:

```
struct A {
    template<typename U> struct B {};

    // template<> // Would be ill-formed by ISO C++.
    //     struct B<int> {};
};
template<> struct A::B<int> {}; // Ok.
```

There is an additional limitation with member class templates of class templates. As with members of a non-template class, one cannot write a specialization at class scope. Example:

```
template<typename T> struct G {
    template<typename U> struct H {};
    // template <> // Would be ill-formed by ISO C++
    //     struct H<int> {};
};
```

But the language specification does not even allow this to be expressed in a namespace-scope definition; there is no way to write an explicit specialization that is a member of a class template. Example:

```
template<typename T> struct J {
    template<typename U> struct K {};
};
// template<typename T>
//     template<> // Would be ill-formed by ISO C++;
//         struct J<T>::K<int> {};
```

This is because the rules for explicit specializations say that '`template<>`' is not allowed to appear after a non-empty template-parameter-list within the same declaration. However, one may write an explicit specialization that is a member of an implicitly-instantiated specialization of a class template. Example:

```
template<typename T> struct L {
    template<typename U> struct M {};
};
template<> template <> struct L<char>::M<int> {}; // Ok
```

Here, the body of the class `L<char>` is automatically generated by implicit instantiation (otherwise the reference to `L<char>::M` would be ill-formed), while the body of `L<char>::M<int>` is provided in the explicit specialization.

In March of 2009, the ISO C++ committee reviewed a report submitted against this example:

```
struct A {
  template<class T> struct B;
  template<class T> struct B<T*> { };  // well-formed
  template<> struct B<int*> { };       // ill-formed
};
```

While it might seem odd that one is able to write the partial specialization but not the full specialization, the committee (which at the time was in a "feature-freeze" mode and trying to finalize a draft for the next International Standard) decided that this capability would need to be regarded as an "extension", meaning that it could be considered as a new feature in a future standard but not as a bug-fix for C++0x.

Note that the Microsoft compiler implements this extension. For that reason, the option

```
-elib(1111)
```

appears in recent versions of our configuration files for Microsoft compilers.

1112 **function with trailing return type must specify return type 'auto', not *string***
error or example, if you want to declare that `f` returns a pointer-to-int, you must write:

```
auto f() -> int *;
```

... and not:

```
auto *f() -> int;
```

This also applies to a `type-id` (e.g., in a cast to a pointer-to-function, or as an argument to a template type-parameter).

1116 **virtual function *symbol* overrides function marked with '*string*'**
error A derived class attempted to override a virtual function that is marked with the `final` virt-specifier in a base class.

1117 **non-virtual function *symbol* marked with '*string*'**
error A virt-specifier (`final` or `override`) was supplied to a non-virtual function.

1118 **virtual function already marked '*string*'**
error A virt-specifier (`final` or `override`) was encountered multiple times for the specified virtual function.

1119 **virtual function *symbol* marked 'override' does not override any member functions**
error A virtual function was marked with the `override` keyword but does not override a base class function.

1120 **incomplete type *type* is not a valid range expression**
error An incomplete type was used as a range expression in a range-based `for` statement. A range expression must be a complete type.

1121 **no viable '*string*' function available for range expression of type *type***
error A non-array range expression used in a range-based `for` statement has no viable `begin` or `end` function.

1122 **range expression of type *type* has '*string*' member but no '*string*' member**
error The type of a range expression used in a range-based `for` statement has either a `begin` or `end` member function but not both.

1123 **attempt to derive from class *type* marked as '*final/sealed*'**
error A class that was marked with the `final` class-virt-specifier was used as a base class in a class declaration.

1124 **digit separator not allowed: *before/after* digit sequence**
error A digit separator character was encountered within a numeric literal at a point where digit separators are not allowed. Digit separators are only allowed between digits of a numeric literal and cannot be adjacent to each other.

1125 **a type cannot be defined in a friend declaration**
error The definition of a type appeared in a friend declaration, this is not legal.

Example:

```
class A {
    friend struct B;    // ok
    friend struct C {}; // error
};
```

1127 **catch handler after catch(...)**
error A `catch` handler appeared following a `catch(...)` in the same `try-catch` statement, which invokes undefined behavior.
Supports MISRA C++ Rule 15-3-7 (Req)

1301 **integer sequences must have non-negative sequence length**
error An attempt was made to instantiate the built-in template `__make_integer_seq` with a negative length.

1302 **integer sequences must have integral element type**
error An attempt was made to instantiate the built-in template `__make_integer_seq` with a non-integral type. As the name implies, `__make_integer_seq` generates a sequence of *integers*.

1401 **non-static data member *symbol* not initialized by constructor *symbol***
warning The indicated non-static data member was not initialized by the specified constructor. Specifically, this means that the member does not have an in-class initializer and was neither directly initialized or assigned a value in the constructor nor did the constructor call any function that appeared to initialize this member.

1404 **deleting an object of type *type* before that type is defined**
warning The following situation was detected:

```
class X; ... X *p; ... delete p;
```

That is, a placeholder declaration for a class is given and an object of that type is deleted before any definition is seen. This may or may not be followed by the actual class definition:

```
class X { ... };
```

A `delete` before the class is defined as dangerous because, among other things, any `operator delete` that may be defined within the class could be ignored.

1405 **header <typeinfo> must be included before 'typeid' is used**
warning According to Section 5.2.8 (para 6) of the C++ standard [11], "If the header `<typeinfo>` (18.5.1) is not included prior to a use of `typeid`, the program is ill-formed." A `typeid` was found in the program but the required include was not.

1407 **incrementing expression of type bool**
warning An increment operator was applied to an object of `bool` type; such use has been deprecated since C++98. The same effect can be obtained by assigning the value `true` to the object. Note the decrementing an object of `bool` type has never been allowed in Standard C++ and will instead be greeted with an error.

1411 **member *symbol* with different signature hides virtual member *symbol***
warning A member function has the same name as a virtual member of a derived class but it has a different signature (different parameter list). This is legal but suspicious because it looks as though the function would override the virtual function but doesn't. You should either adjust the parameters of the member so that the signatures conform or choose a different name. See also message 1511.
Supports MISRA C++ Rule 2-10-2 (Req)

1413 **function *symbol* is returning a temporary via a reference**
warning It appears that a function (identified as *symbol* in the message) declared to return a reference is returning a temporary. The C++ standard (Section 12.2), in addressing the issue of binding temporary values to references, says "A temporary bound to the returned value in a function return statement ... persists until the function exits". Thus the information being returned is not guaranteed to last longer than the function being called.

It would probably be better to return by value rather than reference. Alternatively, you may return a static variable by reference. This will have validity at least until the next call upon the same function.

1414 **assigning address of local variable *symbol* to member of 'this' object**
warning The address of an auto variable was taken and assigned to a `this` member in a member function. For example:

```
struct A {
    char *x;
    void f() {
        char y[10];
```

```
            x = y;       // warning 1414
        }
    };
```

Here, the address of y is being passed to member x but this is dangerous (if not ridiculous) since when the function returns the storage allocated for y is deallocated and the pointer could very easily harm something.

1415 **warning** **pointer to non-POD *type* passed to function *symbol* (*context*)**

A non-POD class is one that goes beyond containing just Plain Old Data (POD). In particular, it may have private or protected data or it may have constructors or a destructor or a copy assignment. All of these things disqualify it from being a POD. A POD is fully defined in the C++ standard (Clause 9).

Some functions (such as memcpy, memcmp, memmove, etc.) are expected to be given only pointers to POD objects. The reason is that only POD objects have the property that they can be copied to an array of bytes and back again with a guarantee that they will retain their original value. (See Section 3.9 of the C++ standard [12]). See also Semantic pod(*i*).

1416 **warning** **reference *symbol* is not yet bound to a value when used here**

This message is usually issued when a reference to a member of a class is used to initialize a reference to another member of the same class before the first member was initialized. For example:

```
    class C {
        int &n, &m;
        C(int &k) : n(m), m(k) { /* ... */ }
    };
```

Here m is initialized properly to be identical to k. However, the initialization of n, taking place, as it does, before m is so initialized, is erroneous. It is undefined what location n will reference.

1417 **warning** ***string* must explicitly initialize the *reference/const* member *symbol***

This message is issued when a reference data member of a class does not appear in a mem-initializer. For example, the following code will result in a Warning 1417 for symbol m since a mem-initializer is the only way that m can be reference initialized.

```
    class C {
        int &n, &m;
        C(int &k) : n(k) { /* ... */ }
    };
```

1419 **warning** **throwing the NULL macro will invoke an implementation-defined handler; NULL may be defined as either an integer literal equal to zero or the keyword nullptr**

The macro NULL was passed to a throw expression. Since C++11, the NULL macro can be defined as expanding to either an integer literal with a zero value or nullptr, the choice of which is implementation defined. The handler that catches the exception may depend on how the NULL macro is defined. Prior to C++11, NULL could only be defined as an integer type and will not be caught by an exception handler expecting a pointer type, which may not be obvious.
Supports MISRA C++ Rule 15-1-2 (Req)

1420 **warning** **'mutable' applied to a reference type is non-standard**

C++ expressly forbids the use of the mutable keyword on a reference type. Despite this, at least one vendor's compiler not only supports this use but relies on the ability to do so in their own library headers. If your compiler supports this use, you can suppress this message.

1421 warning **template parameter illegally redefines default argument**
C++ explicitly forbids redefining the default argument of a template parameter. If your compiler allows this, you can suppress this message.

1501 warning **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 warning **defined object *symbol* of type *symbol* has no non-static data members**
A variable (*symbol*) is being instantiated that belongs to a class (*name*) that contains no data members (either directly or indirectly through inheritance). Note that this message can be suppressed using `-esym` of either the object name or the class name. [10, Section 9]

1504 warning **anonymous structure declaration without the '+fas' option**
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 warning **base specifier with no access specifier is implicitly *public/private***
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. [10, Section 11.1]

1506 warning **call to virtual function *symbol* within a *string***
A call to a virtual function was found in a constructor or a destructor of a class; such a call will not consider overrides from derived classes (as they have not been constructed yet, or have already been destroyed). This message will not be issued in any of the following cases:

- The call was qualified explicitly using the scope operator, inhibiting the virtual call mechanism.
- The virtual function was declared with the final specifier.
- The class of the constructor or destructor was declared with the final specifier.

[14, Section 9]
Supports MISRA C++ Rule 12-1-1 (Req)

1507 warning **use of 'delete' on pointer-to-array type *type* should be 'delete[]'**
The type of an object to be `delete`'d is usually a pointer. This is because operator `new` always returns a

pointer and `delete` may only delete that allocated via `new`. Perhaps this is a programmer error attempting to delete a local or global array? [15]

1509 **the destructor of derived class *type* is non-trivial, but the destructor of base class *type* is not virtual**
warning
The indicated class is a base class for some derived class. It has a non-virtual destructor. Was this a mistake? It is conventional to virtualize destructors of base classes so that it is safe to `delete` a base class pointer. [15]

1510 **the destructor of derived class *type* is non-trivial, but no destructor provided for base class *type***
warning
The indicated class is a base class for some derived class that has a destructor. The base class does not have a destructor. Is this a mistake? The difficulty that you may encounter is this; if you represent (and manipulate) a heterogeneous collection of possibly derived objects via a pointer to the base class then you will need a virtual base class destructor to invoke the derived class destructor. [16, Section 4]

1511 **member function *symbol* hides non-virtual member *symbol***
warning
The named member of a derived class hides a similarly named member of a base class. Moreover, the base class member is not `virtual`. Is this a mistake? Was the base member supposed to have been declared `virtual`? By unnecessarily using the same name, confusion could be created.
Supports MISRA C++ Rule 2-10-2 (Req)

1513 **storage class ignored for member declaration**
warning
A class member was declared with the `extern` or `register` storage class specifier. Member declarations are not allowed to be declared as `extern` or `register`.

1516 **data member *symbol* hides inherited member *symbol***
warning
A data member of a class happens to have the same name as a member of a base class. Was this deliberate? Identical names can cause confusion. To inhibit this message for a particular symbol or for an identifiable set of symbols use `-esym()`.
Supports MISRA C++ Rule 2-10-2 (Req)

1520 **multiple *detail* assignment operators for class *symbol***
warning
More than one assignment operator of the same kind (given in *detail* as `copy`, `move`, or `non-copy non-move`) has been declared for a given class. For example, for class `X` there may have been declared:

```
void operator=(X);
void operator=(X) const;
```

Which is to be used for assignment?

1521 **multiple *copy/move* constructors for class *symbol***
warning
For a given class, more than one function was declared that could serve as a copy or move constructor. Typically, this means that you declared both `X( X& )` and `X( const X& )` for the same class. This is probably a mistake.

1524 **new in constructor for class *symbol* which has no explicit destructor**
warning
A call to `new` has been found in a constructor for a class for which no explicit destructor has been declared.

A destructor was expected because how else can the storage be freed? [11, Section `class.free`]

1526 **undefined member function *symbol***
warning
A member function (named in the message) of a non-library class was not defined. This message is suppressed for unit checkout (`-unit_check` option).

1527 **undefined static data member *symbol***
warning
A static data member (named in the message) of a non-library class was not defined. In addition to its declaration within the class, it must be defined in some module.

1529 **assignment operator *symbol* should check for self-assignment**
warning
The assignment operator does not appear to be checking for assignment of the value of a variable to itself (assignment to `this`). Specifically, PC-lint Plus is looking for the first statement of the assignment operator be an `if` statement which compares `this` to either `&argument` or `std::addressof(argument)` using either `==` or `!=`.
It is important to check for a self assignment so as to know whether the old value should be subject to a delete operation. This is often overlooked by a class designer since it is counter-intuitive to assign to oneself. But, through the magic of aliasing (pointers, references, function arguments) it is possible for an unsuspecting programmer to stumble into a disguised self-assignment. [17, 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 **member allocation function *symbol* of non-final class *symbol* does not reference the allocation size**
warning
A member allocation function (`operator new` or `delete`, including array forms) was declared within a non-final class and does not appear to utilize the dynamic size of the allocation. The size parameter may have been omitted entirely (in the case of `operator delete`) or was never referenced within the function. Because the enclosing class is not final, another class could derive from this class and inherit a member allocation function that relies on a fixed size appropriate only for the base class.

1532 ***operator-delete* should check first parameter for null**
warning
A member `operator delete` should check its argument for NULL as it is unspecified whether deallocation functions are invoked when a null pointer is deleted.

1533 **repeated friend declaration for symbol *symbol***
warning
A `friend` declaration for a particular symbol (class or function) was repeated in the same class. Usually this is a harmless redundancy.

1534 **static variable *symbol* defined within inline function *symbol* in header '*file*'**
warning
A static variable (*symbol*) was found within an inline function within a header file. This can be a source of error since the static variable will not retain the same value across multiple modules. Rather each module will retain its own version of the variable. If multiple modules need to use the function then have the function

refer to an external variable rather than a static variable. Conversely, if only one module needs to use the function then place the definition of the function within the module that requires it. [1, Item 26]

1535 warning **member function *symbol* exposes lower access pointer 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 warning **member function *symbol* exposes lower 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 [4, Item 30]. You may lower the access rights of the function, raise the accessibility of the member, or make the return value a `const` pointer or reference. In the above example you could change the function to:

```
const int *f() { return &a; }
```

Supports MISRA C++ Rule 9-3-1 (Req)
Supports MISRA C++ Rule 9-3-2 (Req)

1537 warning **const member function *symbol* exposes pointer member *symbol* as pointer to non-const**

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. [4, Item 29]

Note, if a `const` function returns the address of a data member then a 605 (capability increase) is issued.

1538 warning **base class *name* absent from initializer list for *copy/move* constructor**

The indicated base class did not appear in the initializer list for a copy or move constructor. Was this an oversight? If the initializer list does not contain an initializer for a base class, the default constructor is used for the base class. This is not normally appropriate for a copy or move constructor. The following is more typical:

```
class B { ... };
class D : public B {
```

```
        D( const D &arg ) : B( arg ) { ... }
        ...
    };
```

1539 warning **member *symbol* not assigned by assignment operator *symbol***
The indicated *symbol* was not assigned by an assignment operator. Was this an oversight? It is not strictly necessary to initialize all members in an assignment operator because the '`this`' class is presumably already initialized. But it is easy to overlook the assignment of individual members. It is also easy to overlook your responsibility to assign base class members. This is not done for you automatically. [4, 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 warning **non-static pointer data member *symbol* not deallocated nor zeroed by destructor *symbol***
The indicated member is a non-static pointer member of a class that was apparently not freed by the class destructor. Was this an oversight? By freeing, we mean either a call to the `free()` function or use of the `delete` operator. If the pointer is intended only to point to static information during its lifetime then, of course, it never should be freed. In that case you should signal closure by assigning it the NULL pointer (`0`).

1541 warning **non-static data member *symbol* possibly not initialized by constructor *symbol***
The indicated non-static data member may not have been initialized by the specified constructor. Specifically, this means that the member does not have an in-class initializer, was not present in the member-initializer list, and was assigned a value (directly or indirectly by a called function) in only some of the paths that the constructor takes.

1544 warning **value of variable *symbol* is indeterminate due to run time initialization of *symbol***
A variable (identified by *symbol*) was used in the run-time initialization of a static variable. However this variable itself was initialized at run-time. Since the order of initialization cannot be predicted this is the source of a possible error.

Whereas addresses are completely known at initialization time, values may not be. Whether the value or merely the address of a variable is used in the initialization of a second variable is not an easy thing to determine when an argument is passed by reference or via pointer. For example,

```
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`.

1546 throw outside try in destructor body
warning
The body of a destructor (signature provided within the message) contains a `throw` not within a `try` block. This is dangerous because destructors are themselves triggered by exceptions in sometimes unpredictable ways. The result can be a perpetual loop. [1, Item 11]
Supports MISRA C++ Rule 15-3-1 (Req)
Supports MISRA C++ Rule 15-5-1 (Req)
Supports MISRA C++ Rule 15-5-3 (Req)

1547 assignment of array of derived class to pointer to base class (*context*)
warning
An assignment from an array of a derived class to a pointer to a base class was detected. For example:

```
class B { };
class D : public B {};
D a[10];
B *p = a;          // Warning 1547
B *q = &a [0];     //OK
```

In this example `p` is being assigned the address of the first element of an array. This is fraught with danger since access to any element other than the zeroeth must be considered an error (we presume that `B` and `D` actually have or have the potential to have different sizes). [1, Item 3]

We do not warn about the assignment to `q` because it appears that the programmer realizes the situation and wishes to confine `q` to the base object of the zeroeth element of a only. As a further precaution against inappropriate array access, out of bounds warnings are issued for subsequent references to `p[1]` and `q[1]`.

1548 exception specification in declaration does not match previous declaration
warning
The exception specification of a function begins with the keyword '`throw`' and follows the prototype. Two declarations were found for the same function with inconsistent exception specifications.
Supports MISRA C++ Rule 15-4-1 (Req)

1549 exception of type *type* thrown from function *symbol* is not in throw list
warning
An exception was thrown (i.e., a `throw` was detected) within a function and not within a `try` block; more over the function contains a `throw` specification but the exception thrown was not on the list. If you provide an exception specification, include all the exception types you potentially will throw. [18, Item 14]
Supports MISRA C++ Rule 15-5-2 (Req)

1550 exception '*name*' thrown by function *symbol* is not on throw-list of function *symbol*
warning
A function was called (first *symbol*) that was declared as throwing an exception. The call was not made from within a `try` block and the function making the call (second *symbol*) had an exception specification that did not include one of the types specified in the called function's exception specification. Either add the exception to the calling function's exception list, or place the call inside a `try` block and catch the `throw`. [1, Item 14]

1551 function *symbol* called outside of a try block in destructor *symbol* is not declared as never throwing
warning
A call to a function (given by the first *symbol*) was made from within a destructor given by the second *symbol*. The called function was declared as potentially throwing an exception. Such exceptions need to be caught within a `try` block because destructors should never throw exceptions. [1, Item 11].

1552 **converting pointer to array of derived to pointer to base**
warning
This warning is similar to Warning 1547 and is sometimes given in conjunction with it. It uses value tracking to determine that an array (that could be dynamically allocated) is being assigned to a base class pointer. For example,

```
Derived *d = new Derived[10];
Base *b;
b = d;       // Warning 1552
b = &d[0];   //OK
```

This case is an issue because if one tries to access `b[i]`, where `i` is an index value, the compiler will attempt to access the object with the address `i * sizeof(Base)` from `b`. However, since the size of `Derived` is almost certain to be larger than the size of `Base`, this object is not the `i`-th `Derived` object.

[1, Item 3] Also, see the article by Mark Nelson (Bug++ of the Month, Windows Developer's Journal, May 1997, pp. 43-44).

1554 **shallow pointer copy of *symbol* in copy constructor *symbol***
warning
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 **shallow pointer copy of *symbol* in copy assignment operator *symbol***
warning
In a copy assignment operator a pointer was merely copied rather than recreated with new storage. This can create a situation where two objects have the same data and this, in turn, causes problems when these objects are deleted or modified. For example, the following class will draw this warning:

```
class X {
    char *p;
    X &operator=(const X &x) { p = x.p; }
};
```

Here, member `p` is expected to be recreated using `new` or some variant.

1556 **initialization could be confused with array allocation**
warning
A `new` expression had the form `new` T`(`*integer*`)` where type T has no constructor. For example:

```
new int(10);
```

will draw this warning. The expression allocates an area of storage large enough to hold one integer. It then initializes that integer to the value `10`. Could this have been a botched attempt to allocate an array of 10 integers? Even if it was a deliberate attempt to allocate and initialize a single integer, a casual inspection of the code could easily lead a reader astray.

The warning is only given when the type T has no constructor. If T has a constructor then either a syntactic error will result because no constructor matches the argument or a match will be found. In the latter case no warning will or should be issued.

1558 warning **inline virtual function is unusual**

The function declared both `virtual` and `inline` has been detected. An example of such a situation is as follows:

```
class C {
    virtual inline void f();
};
```

Virtual functions by their nature require an address and so inlining such a function seems contradictory. We recommend that the `inline` function specifier be removed.

1559 warning **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. [1, Item 11]

Supports MISRA C++ Rule 15-5-1 (Req)

Supports MISRA C++ Rule 15-5-3 (Req)

1560 warning **uncaught exception '*name*' not on throw-list of function *symbol***

A direct or indirect `throw` of the named exception occurred within a `try` block and was either not caught by any handler or was rethrown by the handler. Moreover, the function has an exception specification and the uncaught exception is not on the list. Note that a function that fails to declare a list of thrown exceptions is assumed to potentially throw any exception.

Supports MISRA C++ Rule 15-3-4 (Req)

1562 warning **exception specification for *symbol* is not a subset of *symbol***

The first *symbol* is that of an overriding virtual function for the second *symbol*. The exception specification for the first was found not to be a subset of the second. For example, it may be reasonable to have:

```
struct B   { virtual void f() throw(B); };
struct D:B { virtual void f() throw(D); };
```

Here, although the exception specifications are not identical, the exception `D` is considered a subset of the base class `B`.

It would not be reasonable for `D::f()` to throw an exception outside the range of those thrown by `B::f()` because in general the compiler will only see calls to `B::f()` and it should be possible for the compiler to deduce what exceptions could be thrown by examining the static call.

1563 warning **unparenthesized assignment as false expression of conditional operator**

The third argument to `?:` contained an unparenthesized assignment operator such as

```
p ? a : b = 1
```

If this is what was intended you should parenthesize the third argument as in:

```
p ? a : (b = 1)
```

Not only is the original form difficult to read but C, as opposed to C++, would parse this as:

```
(p ? a : b) = 1
```

1564 warning **converting integer constant expression, which evaluates to *integer* but is not an integer literal equal to zero or one, to bool**

The following looks suspicious.

```
bool a = 34;
```

Although there is an implicit conversion from integral to `bool` and assigning an integer variable to a `bool` to obtain its Boolean meaning is legitimate, assigning an integer such as this looks suspicious. As the message suggests, the warning is not given if the value assigned is either `0` or `1`. An Elective Note would be raised in that instance.

1565 **non-static data member *symbol* not initialized by initializer function *symbol***
warning A function dubbed '`initializer`' by a `-sem` option is not initializing (i.e., assigning to) every data member of a class. `const` members theoretically can be initialized only via the constructor so that these members are not candidates for this message.

1566 **non-static data member *symbol* may have been initialized in a separate method but no '-sem(*name*,initializer)' option was seen**
warning A class data member (whose name and location are indicated in the message) was not directly initialized by a constructor. It may have been initialized by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint Plus that the separately called function is in fact an '`initializer`'. For example:

```
class A {
    int a;
public:
    void f();
    A() { f(); }
};
```

Here `f()` is presumably serving as an initializer for the constructor `A::A()`. To inform PC-lint Plus of this situation, use the option:

```
-sem( A::f, initializer )
```

This will suppress Warning `1566` for any constructor of `class A` that calls `A::f`.

1570 **binding reference field *symbol* to non-reference parameter *symbol***
warning In a constructor initializer, a reference class member is being initialized to bind to an auto variable. Consider:

```
class X { int &n; X(int k) :n(k) {} };
```

In this example member `n` is being bound to variable `k`, which although a parameter, is nonetheless placed into `auto` storage. But the lifetime of `k` is only the duration of the call to the constructor, whereas the lifetime of `n` is the lifetime of the class object constructed.

1571 **returning an auto variable *symbol* via a reference type**
warning A function that is declared to return a reference is returning an `auto` variable (that is not itself a reference). The `auto` variable is not guaranteed to exist beyond the lifetime of the function. This can result in unreliable and unpredictable behavior.

1572 **initializing a static reference variable with an auto variable *symbol***
warning A `static` variable has a lifetime that will exceed that of the `auto` variable that it has been bound to. Consider

```
void f( int n ) { static int& r = n; ... }
```

The reference `r` will be permanently bound to an `auto` variable `n`. The lifetime of `n` will not extend beyond the life of the function. On the second and subsequent calls to function `f` the static variable `r` will be bound

to a non-existent entity.
Supports MISRA C++ Rule 15-3-5 (Req)

1576 **explicit specialization is not in the same file as specialized function template *symbol***
warning
An explicit specialization of a function template was found to be declared in a file other than the one in which the corresponding function template is declared. Two identical calls in two different modules on the same function template could then have two differing interpretations based on the inclusion of header files. The result is undefined behavior.

As if this wasn't enough, if the explicit specialization could match two separate function templates then the result you obtain could depend upon which function templates are in scope.

See also the next message.
Supports MISRA C++ Rule 14-7-3 (Req)

1577 **partial or explicit specialization is not in the same file as specialized class template *symbol***
warning
There is a danger in declaring an explicit specialization or a partial specialization in a file other than that which holds the primary class template. The reason is that a given implicit specialization will differ depending on what headers it sees. It can easily differ from module to module and undefined behavior can be the result.

See also Warning 1576, which diagnoses a similar problem with function templates.
Supports MISRA C++ Rule 14-7-3 (Req)

1578 **non-static pointer data member *symbol* not deallocated nor zeroed by cleanup function *symbol***
warning
The indicated member is a non-static data member of a class that was apparently not cleared by a function that had previously been given the `cleanup` semantic. By clearing we mean that the pointer was either zeroed or the storage associated with the pointer released via the `free` function or its semantic equivalent or some form of `delete`. See also Warning 1540.

1579 **non-static pointer data member *symbol* may have been zeroed or freed in a separate method but no '-sem(*name*,cleanup)' option was seen**
warning
A class data member (whose name and location are indicated in the message) was not directly freed by the class destructor. There was a chance that it was cleared by a separately called member function. If this is the case you may follow the advice given in the message and use a semantic option to inform PC-lint Plus that the separately called function is in fact a 'cleanup' function. For example:

```
class A {
        int *p;
    public:
        void release_ptrs();
        ~A() { release_ptrs(); }
        };
```

Here `release_ptrs()` is presumably serving as a `cleanup` function for the destructor `~A::A()`. To inform PC-lint Plus of this situation, use the option:

```
-sem( A::release_ptrs, cleanup )
```

A separate message (Warning 1578) will be issued if the `cleanup` function fails to clear all pointers. See also Warning 1566.

1705 info **static member *symbol* could be accessed using a nested name specifier instead of applying operator *string* to an instance of class *symbol***

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. [11, Section `class.static`]

1706 info **extra qualification on member *symbol* 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. [10, Section 9.2]

1707 info **static assumed for *symbol***

`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). [10, Section 12.5]

1710 info **missing 'typename' prior to dependent type name '*string*'**

This message is issued when the standard requires the use of '*typename*' to disambiguate the syntax within a template where it may not be clear that a name is the name of a type or some non-type. (See C++ Standard [11], Section `temp.res`, Para 2). Consider:

```
template <class T>
class A {
    T::N x;   // Info 1710
};
```

while technically ill-formed, some compilers will accept this construct since the only interpretation consistent with valid syntax is that `T::N` represents a type. (But if the 'x' weren't there it would be taken as an access declaration and more frequently would be a non-type.)

1711 info **class *symbol* 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. [16, section 4]

1714 info **member function *symbol* not referenced**

A member function was not referenced. This message is automatically suppressed for unit checkout (`-unit_check`) and for members of a library class.
Supports MISRA C++ Rule 0-1-10 (Req)

1715 **static member *symbol* not referenced**
info A static data member of a class was not referenced. This message is automatically suppressed for unit checkout (`-unit_check`) and for members of a library class.
Supports MISRA C++ Rule 0-1-3 (Req)

1719 **reference parameter of copy assignment operator *symbol* should be *type***
info The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not a reference then your program is subject to implicit function calls and less efficient operation. [10, Section 13.4.3]

1720 **reference parameter of copy assignment operator *symbol* should be *type***
info The typical assignment operator for a class is of the form:

```
X& operator =(const X &)
```

If the argument is not `const` then your program will not be diagnosed as completely as it might otherwise be. [10, Section 13.4.3]

1721 **operator=() for class *symbol* is not a copy nor move assignment operator**
info Class assignment operators typically have one of the following forms:

```
X& operator=(const X &);  // copy assignment
X& operator=(X &&);       // move assignment
```

A member function whose name is `operator=` but does not have one of these forms is unusual and may be a subtle source of bugs. If this is not an error you may selectively suppress this message for the given class.
Supports MISRA C++ Rule 14-5-3 (Req)

1722 **assignment operator *symbol* should return *type***
info The typical assignment operator for a class `X` is of the form:

```
X& operator =(const X &);
```

The reason for returning a reference to class is to support multiple assignment as in:

```
a = b = c
```

See also messages `9409` and `9412`.

[10, Section 13.4.3]

1724 **parameter of copy constructor for class *symbol* should be a const reference**
info The parameter for a copy constructor is generally declared as a reference to `const`. This signature is not only standard practice but is also the way a compiler-provided copy constructor is generated. Using a reference that is not to `const` will prevent it from accepting rvalues, including temporary objects (although an applicable move constructor would take precedence if available). The omission of `const` can also cause unexpected results when a copy constructor is declared as deleted, for example:

```
struct A {
    A(int) { }
    operator bool() { }
    A(A&) = delete; // should be const A&
```

```
};
void f() {
    A b = A(5); // compiles even though the copy constructor is deleted
}
```

If the copy constructor in this example had been declared to take a reference to `const`, the temporary would have triggered the selection of the copy constructor and led to the compilation error that would probably be expected when attempting an operation resembling copy construction. Without `const`, the compiler does not consider the copy constructor because the parameter type cannot accept an xvalue, and instead a circuitous and perhaps unexpected conversion sequence is chosen.

1726 info **deletion of pointer to const function parameter *symbol***
The delete operator was applied to either a pointer to const or an array of const. While permitted by the C++ standard, the practice is questioned. If a function didn't have the capability of writing into the area designated by a pointer why would we suppose it to be ok to delete the area?

1727 info **function *symbol* declared inline here was not previously declared inline**
A function declared or defined inline was not previously declared inline. Was this intended? If this is your standard practice then suppress this message. [10, Section 9.3.2]

1728 info **function *symbol* was previously declared inline**
A function was previously declared or defined inline. The inline modifier is absent from the current declaration or definition. Was this intended? If this is your standard practice then suppress this message. [10, Section 9.3.2]

1729 info **initializer inversion: *field/base class symbol* appears before, but will be initialized after, *field/base symbol* in member initializer list**
In a constructor initializer the order of evaluation is determined by the member order not the order in which the initializers are given. At least one of the initializers was given out of order. Was there a reason for this? Did the programmer think that by changing the order that he/she would affect the order of evaluation? Place the initializers in the order of their occurrence within the class so that there can be no mistaken assumptions. [4, Item 13]

1730 info ***string string type* was previously declared as a *string string***
An object is declared both with the keyword `class` and with the keyword `struct`. Though this is legal it is suspect. [10, Section 7.1.6]

1731 info **public virtual function *symbol***
A class member function was declared both `public` and `virtual`. Some authors, see [8, Rule 39], have advocated avoiding public virtual functions because such functions are both part of the public interface and a customization point, aspects often with conflicting motives and audiences. Rather than make the virtual function public consider making it protected. This way members of the hierarchy may still customize behavior.

1732 info **new in constructor for class *symbol* which has no user-provided copy 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. [4, Item 11]

1733 info **new in constructor for class *symbol* which has no user-provided 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. [4, Item 11]

1735 info **parameter '*string*' of virtual function *symbol* has a default argument**
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). [4, Item 38].
Supports MISRA C++ Rule 8-3-1 (Req)

1736 info **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:
    ...
public:
    ...
}
```

In general there are no compiler warnings that would result from such an unintentional botch.

1738 info ***copy/move* constructor for class *symbol* explicitly invokes non-*copy/move* constructor for base class *symbol***
In an initializer list for a copy constructor, a base class constructor was invoked. However, this base class constructor was not itself a copy constructor. We expect that copy constructors will invoke copy constructors. Was this an oversight or was there some good reason for choosing a different kind of constructor? If this was deliberate, suppress this message. See also message `1538`.

1746 info **parameter *symbol* of function *symbol* could be 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 may be more efficient since less information needs to be placed onto the stack and a constructor need not be called.

The message is only given with class-like arguments (including `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. [4, Item 22].

This message is not issued for `extern "C"` functions, which presumably cannot employ references.

1747 info **binary operator *symbol* returns reference type *type***
An operator-like function was found to be returning a reference. For example:

```
X &operator+ ( X &, X & );
```

This is almost always a bad idea. [4, 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 info **non-virtual base class *symbol* included twice in class *symbol***
Through indirect means, a given class was included at least twice as a base class for another class. At least one of these is not virtual. Although legal, this may be an oversight. Such base classes are usually marked `virtual` resulting in one rather than two separate instances of the base class. This is done for two reasons. First, it saves memory; second, references to members of such a base class will not be ambiguous.
Supports MISRA C++ Rule 10-1-3 (Req)

1749 info **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'. [1, Item 24]. The message is inhibited if unit checkout (`-unit_check`) is selected.

1751 info **anonymous namespace declared in a header file**
An unnamed namespace was used in a header file.
Supports MISRA C++ Rule 7-3-3 (Req)

1752 catch parameter is not a reference

info This message is issued for every `catch` parameter that is not a reference and is not numeric. The problem with pointers is a problem of ownership and delete responsibilities; the problem with a non-ref object is the problem of slicing away derivedness [1, Item 23].

Supports MISRA C++ Rule 15-3-5 (Req)

1753 overloading operator *symbol* precludes short-circuit evaluation

info This message is issued whenever an attempt is made to declare one of these operators as having some user-defined meaning:

```
operator ||
operator &&
operator ,
```

The difficulty is that the working semantics of the overloaded operator is bound to be sufficiently different from the built-in operators, as to result in possible confusion on the part of the programmer. With the built-in versions of these operators, evaluation is strictly left-to-right. With the overloaded versions, this is not guaranteed. More critically, with the built-in versions of `&&` and `||`, evaluation of the 2nd argument is conditional upon the result of the first. This will never be true of the overloaded version. [1, Item 7].

Supports MISRA C++ Rule 5-2-11 (Req)

1754 expected symbol '*string*' to be declared for class *symbol*

info The first *symbol* is of the form: `operator` *op*`=` where *op* is a binary operator. A binary operator *op* was declared for type `X` where `X` is identified by the second *symbol*. For example, the appearance of:

```
X operator+( const X \&, const X \& );
```

somewhere in the program would suggest that a `+=` version appear as a member function of class `X`. This is not only to fulfill reasonable expectations on the part of the programmer but also because `operator+=` is likely to be more efficient than `operator+` and because `operator+` can be written in terms of `operator+=`. [1, Item 22]

The message is also given for member binary operators. In all cases the message is not given unless the return value matches the first argument (this is the implicit argument in the case of a member function).

1757 discarded instance of member post-*string* operator

info A postfix increment or postfix decrement operator was used in a context in which the result of the operation was discarded. For example:

```
X a;
...
a++;
```

In such contexts it is just as correct to use prefix decrement/increment. For example this could be replaced with:

```
X a;

...
++a;
```

The prefix form is (or should be) more efficient than the postfix form because, in the case of user-defined types, it should return a reference rather than a value (see 1758 and 1759). This presumes that the side effects of the postfix form are equivalent to those of the prefix form. If this is not the case then either make them equivalent (the preferred choice) or turn this message off. See also 2902, which is issued for non-class types. [1, Item 6].

1758 **prefix *symbol* does not return a reference**
info To conform with most programming expectations, a prefix increment/decrement operator should return a reference. Returning a reference is both more flexible and more efficient [1, Item 6].

The expected form is as shown below:

```
class X {
    X & operator++();
    X operator++( int );
    ...
};
```

1759 **postfix *symbol* returns a reference**
info To conform with most programming expectations, a postfix increment/decrement operator should return a value as opposed to a reference. [1, Item 6]. See example in message 1758.

1762 **member function *symbol* could be made const**
info The indicated (non-static) member function did not modify member data and did not call non-const functions. Moreover, it does not make any deep modification to the class member. A modification is considered deep if it modifies information indirectly through a class member pointer. Therefore, it could and probably should be declared as a `const` member function.
Supports MISRA C++ Rule 9-3-3 (Req)

1763 **const member function *symbol* contains deep modification**
info The designated symbol is a member function declared as `const`. Though technically valid, the `const` may be misleading because the member function modifies (or exposes) information indirectly referenced by the object. For example:

```
class X {
    char *pc;
    char &get(int i) const { return pc[i]; }
};
```

results in Info `1763` for function `X::get`. This is because the function exposes information indirectly held by the `class X`.

Experts [18] 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 [4, Item 29] for a further description.
Supports MISRA C++ Rule 9-3-1 (Req)

1764 **parameter *symbol* of function *symbol* could be reference to const**
info As an example:

```
int f( int & k ) { return k; }
```

can be redeclared as:

```
int f( const int & k ) { return k; }
```

Declaring a parameter a reference to `const` offers advantages that a mere reference does not. In particular, you can pass constants, temporaries and `const` types into such a parameter where otherwise you may not. In addition it can offer better documentation.

Other situations in which a `const` can be added to a declaration are covered in messages 818, 952, 953 and 954.

1766 catch(...) encountered without preceding catch clause

info An ellipsis was used in a catch handler resulting in a handler that will catch any exception. This "catch-all" handler was not preceded by one or more catch handlers in the same try block meaning that this handler will be responsible for processing all exceptions. Catch-all exception handlers are generally considered a bad practice due to the inability to distinguish between different types of exceptions and the potential to hide serious issues. The somewhat less serious use of an exception handler with preceding catch clauses is diagnosed by message 1966.

1768 *access* virtual function *symbol* overrides *access* function in base class *symbol*

info An overriding virtual function has an access (public, protected or private) in the derived class different from the access of the overridden virtual function in the base class. Was this an oversight? Since calls to the overriding virtual function are usually made through the base class, making the access different is unusual (though legal).

1770 function *symbol* defined without function '*string*'

info 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[]`
`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. [1, 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

info This message is given for `operator new` and `operator delete` (and for their `[]` cousins) when a definition for one of these functions is found. Redefining the built-in version of these functions is not considered sound programming practice. [1, Item 23]

1772 assignment operator *symbol* should return *this

info The assignment operator should return `*this`. This is to allow for multiple assignments as in:

```
a = b = c;
```

It is also better to return the object that has just been modified rather than the argument. [4, Item 15]

1773 info **casting away *const/volatile* qualifier without const_cast (*type* to *type*)**
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 [4, Item 21].

1774 info **only dynamic_cast can indicate a failure by returning null – cast from *type* to *type* will not be checked at runtime**
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.
Supports MISRA C++ Rule 5-2-2 (Req)

1775 info **catch block does not catch any declared exceptions**
A catch handler does not seem to catch any exceptions. For example:

```
try { f(); }
catch( B& ) {}
catch( D& ) {}       // Info 1775
catch( ... ) {}
catch( char * ) {}   // Info 1775
```

If `f()` is declared to throw type `D`, and if B is a public base class of `D`, then the first catch handler will process that exception and the second handler will never be used. The fourth handler will also not be used since the third handler will catch all exceptions not caught by the first two.

If `f()` is declared to not throw an exception then Info 1775 will be issued for all four catch handlers.

1776 info **converting string literal to *type* is not const safe**
A string literal, according to Standard C++ is typed an array of `const char`. This message is issued when such a literal is assigned to a non-const pointer. For example:

```
char *p = "string";
```

will trigger this message. This pointer could then be used to modify the string literal and that could produce some very strange behavior.

Such an assignment is legal but "deprecated" by the C++ Standard. The reason for not ruling it illegal is that numerous existing functions have their arguments typed as `char *` and this would break working code.

Note that this message is only given for string literals. If an expression is typed as pointer to const `char` in some way other than via string literal, then an assignment of that pointer to a non-const pointer will receive a more severe warning.
Supports MISRA C 2012 Rule 7.4 (Req)

1777 template recursion limit (*integer*) reached, use -tr_limit(n)
info It is possible to write a recursive template that will contain a recursive invocation without an escape clause. For example:

```
template <class T> class A { A< A > x; };
A<int> a;
```

This will result in attempts to instantiate:

```
A<int>
A<A<int>>
A<A<A<int>>>
...
```

Using the `-vt` option (turning on template verbosity) you will see the sequence in action. Accordingly, we have devised a scheme to break the recursion when an arbitrary depth of recursion has been reached (at this writing 75). This depth is reported in the message. As the message suggests, this limit can be adjusted so that it equals some other value.

When recursion is broken, a complete type is not used in the definition of the last specialization in the list but processing goes on.

1778 assignment of string literal to variable *symbol* is not const safe
info This message is issued when a string literal is assigned to a variable whose type is a non-const pointer. For example:

```
char *p;  p = "abc";
```

The message is issued automatically (i.e. by default) for C++. For C, to obtain the message, you need to enable the Strings-are-Const flag (`+fsc`). This message is similar to message `1776` except that it is issued whenever a string constant is being assigned to a named destination.
Supports MISRA C 2012 Rule 7.4 (Req)

1780 returning address of reference to a const parameter *symbol*
info The address of a parameter that has been declared as being a reference to a `const` is being returned from a function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
const int *f( const int & n )
    { return &n; }
int g();
const int *p = f( g() );
```

Here, p points to a temporary value whose duration is not guaranteed. If the reference is not `const` then you will get Elective Note 1940.

This is an example of the Linton Convention as described by Murray [19].
Supports MISRA C++ Rule 7-5-3 (Req)

1781 **passing address of const reference parameter *symbol* into caller address space**
info
The address of a parameter that has been declared as being a reference to a `const` is being assigned to a place outside the function. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
void f( const int & n, const int **pp )
    { *pp = &n; }
int g();
const int *p;
... f( g(), &p );
```

Here, p will be made to point to a temporary value whose duration is not guaranteed. If the reference is not `const` then you will get Elective Note 1940.

This is an example of the Linton Convention as described by Murray [19].

1782 **assigning address of const reference parameter *symbol* to a static variable**
info
The address of a parameter that has been declared as being a reference to a const is being assigned to a static variable. The danger of this is that the reference may designate a temporary variable that will not persist long after the call. For example:

```
const int *p;
void f( const int & n )
    { p = &n; }
int g();
... f( g() );
```

Here, p will be made to point to a temporary value whose duration is not guaranteed. If the reference is not `const` then you will get Elective Note 1940.

This is an example of the Linton Convention as described by Murray [19].

1784 **symbol *symbol* previously declared as "C"**
info
A *symbol* is being redeclared in the same module. Whereas earlier it had been declared with an `extern "C"` linkage, in the cited declaration no such linkage appears. E.g.

```
extern "C" void f(int);
void f(int);             // Info 1784
```

In this case the `extern "C"` prevails and hence this inconsistency probably represents a benign redeclaration. Check to determine which linkage is most appropriate and amend or remove the declaration in error.

1785 **implicit conversion from Boolean (*context*) (*type* to *type*)**
info
A Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of some other type. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

1786 implicit conversion to Boolean (*context*) (*type* to *type*)
info A non-Boolean expression was assigned (via assignment, return, argument passing or initialization) to an object of type Boolean. Was this the programmer's intent? The use of a cast will prevent this message from being issued.

1787 access declarations are deprecated; use using declarations instead
info The C++ Standard ([11] section 7.3.3) specifically deprecates the use of access declarations. The preferred syntax is the using declaration. For example:

```
class D : public B {
    B::a;         // message 1787
    using B::a;   // preferred form and no message
};
```

In C++11, support for access declarations were removed completely. In C++11 and later modes, this message is replaced with an error.

1788 variable *symbol* of type *symbol* is referenced only by its constructor/destructor
info A variable has not been referenced other than by the constructor that formed its initial value or by its destructor or both. The location of the symbol and also its type is given in the message. For example:

```
struct A {
    A();
};
void f() { A a; }
```

will produce a `1788` for variable '`a`' and for type '`A`'.

It very well may be that this is exactly what the programmer wants to do, in which case you may suppress this message for this variable using the option `-esym(1788,a)`. It may also be that the normal use of `class A` is to employ it in this fashion. That is, to obtain the effects of construction and, possibly, destruction but have no other reference to the variable. In this case the option of choice would be `-esym(1788,A)`.

1789 constructor template *symbol* cannot be a copy constructor
info This message is issued for classes for which a copy constructor was not defined but a template constructor was defined. For example:

```
struct A {
       template <typename T>
       A(const T&);           // Info 1789
};
```

The C++ standard specifically states that a template constructor will not be used as a copy constructor. Hence, a default copy constructor is created for such a class while the programmer may be deluded into thinking that the template will be employed for this purpose. [8, Item 5].
Supports MISRA C++ Rule 14-5-2 (Req)

1790 public base *symbol* of *symbol* has no non-destructor virtual functions
info A public base class contained no virtual functions except possibly virtual destructors. There is a school of thought that public inheritance should only be used to interject custom behavior at the event of virtual function calls. To quote from Marshall Cline, "Never inherit publicly to reuse code (in the base class); inherit publicly in order to be reused (by code that uses base objects polymorphically)" [8, Item 22].

1791 returned expression begins on the next line
info
A line is found that ends with a `return` keyword and with no other tokens following. Did the programmer forget to append a semi-colon? The problem with this is that the next expression is then consumed as part of the `return` statement. Your return might be doing more that you thought. For example:

```
void f( int n, int m ) {
    if( n < 0 ) return   // do not print when n is negative
    print( n );
    print( m );
}
```

Assuming `print()` returns `void`, this is entirely legal but is probably not what you intended. Instead of printing `n` and `m`, for `n` not negative you print just `m`. For `n` negative you print `n`.

To avoid this problem always follow the `return` keyword with something on the same line. It could be a semi-colon, an expression or, for very large expressions, some portion of an expression.

1793 invoking non-const member function *symbol* of class *symbol* on a temporary
info
A non-static and non-const member function was called and an rvalue (a temporary object) of class *symbol* was used to initialize the implicit object parameter. This is legal (and possibly intentional) but suspicious. Consider the following.

```
struct A { void f(); };
...
A().f();           // Info 1793
...
```

In the above the 'non-static, non-const member function' is `A::f()`. The 'implicit object parameter' for the call to `A::f()` is `A()`, a temporary. Since the `A::f()` is non-const it presumably modifies `A()`. But since `A()` is a temporary, any such change is lost. It would at first blush appear to be a mistake.

The Standard normally disallows binding a non-const reference to an rvalue but as a special case allows it for the binding of the implicit object parameter in member function calls. Some popular libraries take advantage of this rule in a legitimate way. For example, the GNU implementation of `std::vector<bool>::operator[]` returns a temporary object of type `std::_Bit_reference` – a class type with a non-const member `operator=()`. `_Bit_reference` serves a dual purpose. If a value is assigned to it, it modifies the original class through its `operator=()`. If a value is extracted from it, it obtains that value from the original class through its operator `bool()`.

This message will not be issued for member functions declared using an rvalue ref-qualifier such as `void f() &&;`.

Probably the best policy to take with this message is to examine instances of it and if this is a library invocation or a specially designed class, then suppress the message with a `-esym()` option.

1795 defined template *symbol* is not instantiated
info
The named template was defined but not instantiated. As such, the template either represents superfluous code or indicates a logic error.

The 'template' in the message could also be a temploid. A temploid is defined as either a template or a member of a temploid.
Supports MISRA C++ Rule 14-7-1 (Req)

1798 **block scope declaration of *symbol* is taken to mean a member of *symbol* but does not introduce a name**
info
A nested function declaration was found within a function whose innermost enclosing namespace was not the global namespace. This alone cannot introduce a namespace member but the declaration of the nested function will still be taken to refer to a (possibly non-existent) member of the innermost enclosing namespace. This can lead to pernicious linker errors if one expects the declared function to introduce a namespace member into the innermost enclosing namespace or the global namespace.

1901 **creating a temporary of type *type***
note
PC-lint Plus judges that a temporary needs to be created. This occurs, typically, when a conversion is required to a user object (i.e. class object). Where temporaries are created, can be an issue of some concern to programmers seeking a better understanding of how their programs are likely to behave. But compilers differ in this regard.

1902 **unnecessary semicolon follows function definition**
note
It is possible to follow a function body with a useless semi-colon. This is not necessarily 'lint' to be removed but may be a preferred style of programming (as semi-colons are placed at the end of other declarations).

1904 **old-style c comment**
note
For the real bridge-burner one can hunt down and remove all instances of the `/*` ... `*/` form of comment. [4, Item 4]

1905 **implicit non-trivial default constructor generated for *symbol***
note
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.

1906 **exception specification for function *symbol***
note
A function was declared with an exception specification. Some authors contend exception specifications are not worth using due to a presumably false sense of security associated with the specifications. See for example [8, Rule 75].

1907 **implicit non-trivial destructor generated for *symbol***
note
The named class does not itself have an explicit destructor but either had a base class that has a destructor or has a member class that has a destructor (or both). In this case a destructor will be generated by the compiler. [10, Section 12.4]

1908 **destructor *symbol* is implicitly virtual due to virtual destructor of base class *symbol* but is not explicitly marked virtual**
note
The destructor cited was inherited from a base class with a virtual destructor. This word '`virtual`' was omitted from the declaration. It is common practice to omit this keyword when implied. See also 1909.

1909 **'virtual' assumed; see function *symbol***
note
The named function overrides a base class virtual function and so is virtual. It is common practice to omit the `virtual` keyword in these cases although some feel that this leads to sloppy programming. This message

allows programmers to detect and make explicit which functions are actually virtual.
Supports MISRA C++ Rule 10-3-2 (Req)

1911 implicit call of converting constructor *symbol*
note The *symbol* in the message is the name of a constructor called to make an implicit conversion. This message can be helpful in tracking down hidden sources of inefficiencies. [10, Section 12.1]

1912 implicit call of conversion function from class *symbol* to type *type*
note A conversion function (one of the form *symbol*`::operator` *type* `()`) was implicitly called. This message can be helpful in tracking down hidden sources of inefficiencies.

1914 default constructor *symbol* not referenced
note A default constructor was not referenced. When a member function of a class is not referenced, you will normally receive an Informational message (`1714`) to that effect. When the member function is the default constructor, however, we give this Elective Note instead.

The rationale for this different treatment lay in the fact that many authors recommend defining a default constructor as a general principle. Therefore, if you are following a modus operandi of not always defining a default constructor you may want to turn on message `1914` instead.
Supports MISRA C++ Rule 0-1-10 (Req)

1915 virtual function *symbol* overrides function *symbol* and is not marked with 'override'
note A virtual function that overrides a base class function was not declared with the override virt-specifier. This message is only emitted for C++11 and higher.

1916 function *symbol* is variadic
note An ellipsis was encountered while processing the prototype of some function declaration. An ellipsis is a way of breaking the typing system of C or `C++`.

1919 *symbol* is not a *copy/move* assignment operator
note 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 reference type *type*
note The ARM [10, Section 5.4] states that reference casts are often 'misguided'. However, too many programs are openly using reference casts to place such casts in the Informational category.

1924 use of c-style cast
note A C-style cast was used in C++ code. This can usually be replaced by one of the newer `C++` casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, or a combination thereof. [1, Item 2].
This message is not issued for casts to void used to discard values.
Supports MISRA C++ Rule 5-2-4 (Req)

1925 symbol *symbol* is public data member
note The indicated *symbol* is a public data member of a class. If the class is introduced with the keyword struct the message is not issued. In some quarters the use of public data members is deprecated. The rationale is that if function calls replace data references in the public interface, the implementation can change without affecting the interface. [4, Item 20]

1926 default constructor implicitly called to initialize field *symbol*
note A member of a class (identified by *symbol*) did not appear in the constructor initialization list. Since it had a default constructor this constructor was implicitly called. Is this what the user intended? Some authorities suggest that all members should appear in the constructor initialization list. [4, Item 12]].

1927 data member *symbol* absent from initializer list for constructor
note A member of a class (identified by *symbol*) did not appear in a constructor initialization list. If the item remains uninitialized through the whole of the constructor, a Warning 1401 is issued. Some authorities suggest that all members should appear in the constructor initialization list. [4, Item 12].

1928 base class *name* absent from initializer list for constructor
note A base class (identified by *symbol*) did not appear in a constructor initialization list. If a constructor does not appear, the default constructor is called. This may or may not be valid behavior. If a base class is missing from the initializer list of a copy constructor (as opposed to some ordinary constructor), a more severe Warning (1538) is issued. [4, Item 12].
Supports MISRA C++ Rule 12-1-2 (Adv)

1929 non-member function *symbol* returns reference type *type*
note A non-member function was found to be returning a reference. This is not normally considered good practice because responsibility for deleting the object is not easily assigned. No warning is issued if the base class has no constructor. [4, Item 23].

1930 conversion operator *symbol* found
note 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 [1, Item 5].

1931 constructor *symbol* can be used for implicit conversions
note A constructor was found that could be used for implicit conversions. For example:

```
class X
    {
  public:
    X(int);
    ...
    };
```

Here any `int` (or type convertible to `int`) could be automatically converted to `X`. This can sometimes cause confusing behavior[1, Item 5]. If this is not what was intended, use the keyword '`explicit`' as in:

```
explicit X(int);
```

This will also serve to suppress this message. See also message 9169.

1932 base class *type* is not abstract
note An abstract class is a class with at least one pure virtual specifier. At least one author has argued [1, Item 33] that all base classes should be abstract although this suggestion flies in the face of existing practice.

1933 call to unqualified virtual function *symbol* from non-static member function
note A classical C++ gotcha is the calling of a virtual function from within a constructor or a destructor. When we discover a direct call from a constructor or destructor to a virtual function we issue Warning 1506. But what about indirect calls. Suppose a constructor calls a function that in turn, perhaps through several levels of call, calls a virtual function. This could be difficult to detect. Dan Saks [18] has suggested a compromise Guideline that "imposes few, if any, practical restrictions". The Guideline, implemented by this Elective Note, issues a message whenever an unqualified virtual function is called by any other (non-static) member function (for the same 'this' object). For example:

```
class X { virtual void f(); void g(); };

void X::g()
    {
    f();          // Note 1933
    X::f();       // ok -- non virtual call.
    }
```

Even if total abstinence is unwarranted, turning on message 1933 occasionally can be helpful in detecting situations when constructors or destructors call virtual functions.

1934 shift operator *symbol* should be a non-member function
note It has been suggested [4, Item 19] that you should never make a shift operator a member function unless you're defining `ostream` or `istream` (the message is suppressed in these two cases). The reason is that there is a temptation on the part of the novice to, for example, define output to `ostream` as a class member function left shift that takes `ostream` as an argument. This is exactly backwards. The shift operator normally employs the destination (or source) on the left.

On the other hand, if the class you are defining is the source or destination then defining the shift operators is entirely appropriate.

1937 static variable *symbol* of type *type* has a non-trivial destructor
note A static scalar whose name is *symbol* has a destructor. Destructors of static objects are invoked in a predictable order only for objects within the same module (the reverse order of construction). For objects in different modules this order is indeterminate. Hence, if the correct operation of a destructor depends on the existence of an object in some other module an indeterminacy could result. See also 1935, 1936, 1544 and 1545.

1938 constructor *symbol* accesses global data
note A constructor is accessing global data. It is generally not a good idea for constructors to access global data because order of initialization dependencies can be created. If the global data is itself initialized in another module and if the constructor is accessed during initialization, a 'race' condition is established. [4, Item 47]
Supports MISRA C++ Rule 12-8-1 (Req)

1939 **casting from base class *type* to derived class *type***
note A down cast is a cast from a pointer (or reference) to a base class to a pointer (or reference) to a derived class. A cast down the class hierarchy is fraught with danger. Are you sure that the alleged base class pointer really points to an object in the derived class? Some amount of down casting is necessary, but a wise programmer will reduce this to a minimum. [4, Item 39]
Supports MISRA C++ Rule 5-2-2 (Req)

1940 **address of non-const reference parameter *symbol* transferred outside of function**
note The address of a reference parameter is being transferred (either via a `return` statement, assigned to a static, or assigned through a pointer parameter) to a point where it can persist beyond the lifetime of the function. These are all violations of the Linton Convention (see Murray [19]).

The particular instance at hand is with a reference to a non-`const` and, as such, it is not considered as dangerous as with a reference to a `const`. (See 1780, 1781 and 1782 for those cases). For example:

```
int *f( int &n ) { return &n; }
int g();
int *p = f( g() );
```

would create a problem were it not for the fact that this is diagnosed as a non-`lvalue` being assigned to a reference to non-`const`.
Supports MISRA C++ Rule 7-5-3 (Req)

1941 ***string* assignment operator *symbol* does not return *type***
note The typical use of an assignment operator for class `C` is to assign new information to variables of class `C`. If this were the entire story there would be no need for the assignment operator to return anything. However, it is conventional to support chains of assignment as in:

```
C x, y, z;
...
x = y = z;
// parsed as x = (y = z);
```

For this reason assignment normally returns a reference to the object assigned the value. For example, assignment `(y = z)` would return a reference to `y`.

Since it is almost never the case that this variable is to be reassigned, i.e. we almost never wish to write:

```
(x = y) = z; // unusual
```

as a general rule it is better to make the assignment operator return a `const` reference. This will generate a warning when the unusual case is attempted.

But experts differ. Some maintain that in order to support non-const member functions operating directly on the result of an assignment as in:

```
(x = y).mangle();
```

where, as its name suggests, `mangle` is non-const it would be necessary for the return value of assignment to be non-const. Another reason to not insist on the `const` qualifier is that the default assignment operator returns simply a reference to object and not a reference to `const` object. In an age of generic programming, compatibility may be more important than the additional protection that the `const` would offer.

1943 **declaration of *symbol* of type *type* may require global runtime construction**
note This message is issued for file-scope variables of class type that have a non-trivial constructor that requires

the constructor to be executed to initialize the object at startup time. This can be a potential performance concern.

1944 declaration of *symbol* of type *type* requires a global destructor
note This message is issued for file-scope variables of class type that have a non-trivial destructor that requires the destructor to be executed to destroy the object at shutdown time. This can be a potential performance concern.

1945 declaration of *symbol* of type *type* requires an exit-time destructor
note This message is issued for file-scope variables of class type that have a non-trivial destructor that requires the destructor to be executed to destroy the object at shutdown time. This can be a potential performance concern.

1962 member function *symbol* contains deep modification
info The designated member function could be declared `const` but shouldn't be because it contains a deep modification. For example:

```
class X {
    char *p;

public:
    void f() { *p = 0; }
};
```

will elicit this message indicating that `X::f()` contains a deep modification. A modification is considered shallow if it modifies (or exposes for modification) a class member directly. A modification is considered deep if it modifies information indirectly through a class member pointer. This Elective Note is available for completeness so that a programmer can find all functions that could result in a class being modified. It does not indicate that the programming is deficient. In particular, if the function is marked `const` an Info 1763 will be issued. See also 1762, 1763.

1966 catch(...) encountered after catch clause
note An ellipsis was used in a catch handler resulting in a handler that will catch any exception. This "catch-all" handler was preceded by one or more catch handlers in the same try block such that this handler will catch any exceptions not caught by one of the more specific handlers. Catch-all exception handlers are generally considered a bad practice due the the inability to distinguish between different types of exceptions and the potential to hide serious issues. The use of such an exception handler without any preceding catch clauses is diagnosed by message 1766.

1970 use of default capture (*string*) in lambda expression
note This Elective Note diagnoses the use of default capture in a lambda expression. *string* is either `=` or `&`. The use of default capture can have unintended consequences, even in apparently innocuous situations and as such it has been suggested that default capture never be used. For an in-depth discussion of the issue, see [20, Item #1].

1971 use of function try block for non-constructor function *symbol*
note The motivation for the creation of the function-try block in C++ is to allow for the handling of exceptions thrown during the processing of constructor initializer lists. Such exceptions cannot be handled inside of the body of the constructor as the body is not yet entered. While function-try blocks are allowed for

non-constructor functions, the same functionality can be obtained using the more general try-catch block inside the body of the function.

1972 **empty declaration**
note An empty declaration was encountered; this can happen from an extraneous semi-colon:

```
int x;;
```

Note: In PC-lint this was reported as error `19`.

1973 **deletion of non-parameter pointer to const**
note The `delete` operator was applied to a non-parameter pointer to const. This is legal and not necessarily suspect. See also message 1726 reports on cases where the pointer being deleted is a function parameter, which is more likely to result in unexpected behavior.

15.4 Messages 2000-2999

2001 **request for *string* integer type with at least *integer* bits could not be processed**
error An appropriately sized integer type could not be found when attempting to determine the smallest integer type with enough bits to represent a bitfield or an integer constant expression.

2006 **no hexadecimal digits following *string* escape sequence**
error A `\x` or `\u` escape sequence was seen but there were no hexadecimal digits immediately following the sequence.

2400 **unexpected internal condition '*string*'**
warning PC-lint Plus has encountered an unexpected situation while processing the provided source code. This doesn't necessarily represent either a bug in the source code or in PC-lint Plus, and PC-lint Plus will continue to operate normally, but rather serves to report potentially interesting circumstances that may be of use to Gimpel Software engineering staff. This message is not emitted unless appropriate debugging options are enabled.

2401 **cannot mix positional and non-positional arguments in format string**
warning The format string for a `printf`/`scanf` style function contains both positional and non-positional arguments. Positional arguments are an extension provided by POSIX implementations but mixing positional and non-positional arguments results in undefined behavior. For example:

```
printf("%1$d %d", 1, 2);
```

will elicit this message.

2402 **'*string*' specified field *string* is missing a matching 'int' argument**
warning The format string for a `printf`/`scanf` style function contains a conversion specifier whose width or precision is given as an asterisk (`*`) indicating that the width/precision be extracted from the next argument, which should have type `int`, but this argument was not provided.

2403 **field *string* should have type *type*, but argument has type *type***
warning The width or precision of a conversion specifier within the format for a `printf` or `scanf` style function was

specified with an asterisk (`*`) and as such a corresponding `int` argument was expected to represent the width/precision but the argument in that position was not the correct type. For example:

```
extern double f;
printf("%*d", f, f);
```

will yield the messages:

```
field width should have type 'int', but argument has type 'double'
```

2404 invalid position specified for *string*
warning
Within the format string of a `printf` or `scanf` style function, a positional parameter specifier was expected for a field width or precision that used the asterisk (`*`) to indicate that the field or width should be taken from the argument list but one was not provided. For example:

```
printf("%1$*d", 1, 2);
```

will yield the message:

```
invalid position specified for field width
```

This is because when positional specifiers are used within a format string, all arguments must have corresponding positional specifiers. The correct way to indicate that the field width corresponds to the second data argument is:

```
printf("%1$*2$d", 1, 2);
```

2405 *string* used with '*string*' conversion specifier is undefined
warning
The use of a field width or precision with an incompatible conversion specifier has been encountered. Standard C allows a precision to be used only with the `d`, `I`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers and a field width to be used with any conversion specifiers except for `n.` Use of field width/precision outside of these conversion specifiers results in undefined behavior.

2406 no closing ']' for '%[' in scanf format string
warning
Within a format string for a `scanf` style function, a '`%[`' was seen denoting the start of a scan list but there was no terminating '`]`'. The lack of a closing bracket makes the conversion specification invalid and results in undefined behavior.
Supports MISRA C 2004 Rule 4.1 (Req)

2407 zero field width in scanf format string is unused
warning
Within a `scanf` style function, a zero was given as the maximum field width. Standard C specifies that the maximum field width for `scanf` must be a "decimal integer greater than zero". Providing a zero as the width makes the conversion specifier invalid resulting in undefined behavior.

2408 cannot pass *string* object of type *type* to variadic *string*; expected type from format string was *type*
warning
A non-POD or non-trivial class type that cannot be passed as a variadic function argument was given as the argument to a `printf`/`scanf` style function. The first *type* specifies the type of the argument that was provided, the second *type* specifies the type that was expected from the format string.

2409 format string should not be a wide string
warning
The format string for a non-wide version of a `printf`/`scanf` style function was a wide string literal but should instead be an ordinary (narrow) string literal. For example:

```
printf(L"%d", 1);
```

will elicit this message.

2410 **warning** **re-entrant initializer for static local variable *symbol* causes undefined behavior**

Recursively executing the initializer for a static local variable is undefined behavior, even if it appears not to cause an infinite loop. An implementation with proper support for thread-safe static initialization is likely to deadlock.

2423 **warning** **apparent domain error for function *symbol*, argument *integer* (value=*string*) outside of accepted range (*string*)**

A value was provided to a mathematical function that will result in a domain error. For example, the `acos` function is only defined for values in the range [-1, 1], values provided outside this range will be diagnosed by this message. Value tracking is used to determine the value provided to the function. For example:

```
double foo(double x, double y) {
    acos(x + y);
}
void bar() {
    foo(0.5, 0.75);
}
```

will elicit the message:

```
warning 2423: apparent domain error for function 'acos(double)',
    argument 1 (value=1.25) outside of accepted range (between -1 and 1)
acos(x + y);
       ^
```

Supports MISRA C 2012 Directive 4.11 (Req)

2425 **warning** **user-defined function semantic '*string*' was rejected during call to function *symbol* because *string***

A call was made to a function for which a user-defined semantic exists but the semantic could not be applied because it contains a semantic that is not valid for this call. There are several reasons this can occur including specifying a semantic for an argument that does not exist, a return value of a type that conflicts with the actual return value, or the use of a symbol or macro in the semantic that cannot be resolved at the time of the call.

2426 **warning** **return value (*string*) of call to function *symbol* conflicts with return semantic '*string*'**

A user-defined return semantic was specified for a function for which PC-lint Plus has access to the implementation. Furthermore, PC-lint has determined that during a specific call of the function the actual value returned conflicts with the claimed return value in the return semantic. This represents a likely error in either the implementation of the function or the specification of the semantic. See also 9.2.2.2 Return Semantic Validation in the Semantics chapter.

2427 **warning** **initializer_list elements will be destroyed before returning**

The array associated with an initializer list is allocated with a temporary lifetime. The lifetime of the array will not be extended beyond the full expression of a return statement. The returned initializer list will contain dangling pointers. For example:

```
#include <initializer_list>
```

```
std::initializer_list<int> f() {
    return { 1, 2, 3 }; // The memory used to store the array
                        // elements will be freed before returning.
}

void g() {
    auto x = f();
    // Attempting to access the elements of x will read invalid memory.
}
```

2430 **missing whitespace between macro name *name* and definition**
warning
Standard C requires the presence of whitespace between a macro name and its definition for object-like macros. For example:

```
#define MINUS-
```

will elicit:

```
warning 2430: missing whitespace between macro name 'MINUS' and definition
#define MINUS-
             ^
```

Despite the warning, the macro `MINUS` is still defined to `-` so it is safe to suppress this message for legacy code that cannot be changed. The best way to address the warning is to place a space between the macro name and definition:

```
#define MINUS -
```

2431 ***#line/GNU line* directive starting with zero is not interpreted as an octal number**
warning
The line number provided to the `#line` *number* preprocessing directive (and the GNU equivalent `#` *number*) is always interpreted as a decimal number, even when the first digit is a zero. For example, `#line 034` is treated as `#line 34`, not as `#line 28` (the decimal equivalent of octal 34). As such, a `#line` directive with a line number beginning with a zero is suspicious.

2432 **macro *name* used as header guard is followed by a #define of a similar but different macro '*name*'**
warning
Within a construct that appears to be a macro include guard, the name of the macro being checked is similar to, but different from, the name of the macro subsequently defined. For example:

```
#ifndef FOO_INCLUDED
#define FOO_INCLODED
...
#endif
```

will elicit:

```
warning 2432: macro 'FOO_INCLUDED' used as header guard is followed by a
    #define of a similar but different macro 'FOO_INCLODED'
#ifndef FOO_INCLUDED
        ^~~~~~~~~~~~
```

This usually represents a typo, which will prevent the include guard from functioning as intended. This message can be suppressed with `-estring` using the name of the macro being defined, e.g. `-estring(2432, FOO_INCLODED)` if the difference was intentional.

2433 **conversion specifier '*string*' is not allowed for bounds-checked format function**
warning
Bounds-checked format functions are described in Annex K of the C11 standard. The bounds-checked `printf`-like functions forbid the use of the `%n` conversion specifier.

2434 **memory was potentially deallocated**
warning
This message is a less certain variant of 449 and is issued when the deallocation was dependent on conditional execution flow at runtime.

2435 **duplicate '*string*' declaration specifier**
warning
The same declaration specifier was used more than once in the declaration of a symbol. For example:

```
inline inline void foo();
```

will elicit this message. Was this intended? While legal, it is suspect. Other specifiers that will be diagnosed for duplicates include `virtual`, `explicit`, `_Noreturn`, `friend`, and `constexpr`.

2436 **function *symbol* declared 'noreturn' should not return**
warning
A function that was declared as not returning either with the keyword `_Noreturn` or a GCC or C++11 noreturn attribute contained a return statement. Returning from a function designated as not returning invokes undefined behavior.

2437 **indirection of non-volatile null pointer may be optimized out**
warning
An indirection on a non-volatile null pointer was encountered. While this is undefined behavior as far as Standard C is concerned, the programmer may have intended for this to generate a trap condition relying on implementation details but the compiler is likely to simply remove the offending indirection instead. The null pointer should be volatile to indicate to the compiler that it should not be optimized out.

2438 **comparing values of different enumeration types (*type* and *type*)**
warning
The values of two different enumeration types were used in an equality or comparison operation. This is suspect because there is no intrinsic relationship among different enumeration types and as such it usually doesn't make sense to compare them. For example:

```
enum color { RED, GREEN, BLUE };
enum fruit { APPLE, PEAR, MANGO };

void foo(enum color c, enum fruit f) {
    if (c == f) return;     // 2438 issued here
    // ...
}
```

The message is parameterized by the two enumeration types compared.

2439 **lint comment does not contain any options**
warning
A lint comment was encountered that did not contain any lint options, was this a mistake? The comment may be empty or may start with text that does not begin an option. For example:

```
//lint e714 -e715
```

Since `e714` does not start with a `-`, `+`, or `!`, it, along with everything that follows, is assumed to be commentary. In this case `-e714` was probably meant.

2440 ***string* '*string*' in comparison is never null**
warning
The address of a function, array, or variable was directly compared to null. This is suspicious because the address of a function or variable can never be null in well-formed code. Note that this message is not given for null checks of function or object pointers. For example:

```
void foo(int *pi) {
    if (!pi) return;        // Okay
    if (&pi == 0) return;   // 2440
    if (foo != 0) return;   // 2440
}
```

The first *string* parameter is one of 'function', 'array', or 'address of' and the second *string* parameter represents the corresponding function, array, or variable.

2441 ***string* '*string*' used in boolean context is never null**
warning
The address of a function, array, or variable was used in a boolean context. This is suspicious because such an address can never be false. Note that this message is not given for function or object pointers. For example:

```
void foo(int *pi) {
    if (!pi) return;    // Okay
    if (&pi) return;    // 2441
    if (!foo) return;   // 2441
}
```

The first *string* parameter is one of 'function', 'array', or 'address of' and the second *string* parameter represents the corresponding function, array, or variable.

2444 **case value is not in enumeration *type***
warning
The condition of a switch statement has `enum` type but contains a `case` statement with a value that doesn't correspond to any of the enumerators in the `enum`. For example:

```
enum color { RED, GREEN, BLUE };
void foo(enum color c) {
    switch (c) {
    case RED:       // OK
    case RED + 1:   // OK, refers to GREEN
    case 2:         // OK, refers to BLUE
    case 3: ...     // Warning 2444, no member with value 3
    }
}
```

2445 **cast from *type* to *type* increases required alignment from *integer* to *integer***
warning
A cast was made from a pointer to one type to a pointer to a type that has greater alignment requirements than the type pointed to by the original pointer. For example, assuming an alignment requirement of 4 bytes for '`int`' and 8 bytes for '`long double`':

```
void foo(int *pi) {
    long double *pld = (long double *)pi;
}
```

will result in the message:

```
warning 2445: cast from 'int *' to 'long double *' increases
    required alignment from 4 to 8
```

```
long double *pld = (long double *)pi;
                   ^~~~~~~~~~~~~~~~~
```

Accessing the value through the new pointer may invoke undefined behavior if it is not properly aligned. The alignment requirements of fundamental types can be set using the -a option.

The message is parameterized by the types of the pointer before and after the cast and the alignment requirements of the types before and after the cast.

2446 warning **pasting formed '*string*', an invalid preprocessing token**
During a token pasting operation performed by the preprocessor `##` operator, an invalid token was formed. This is illegal even if the result is immediately pasted with another token that would then form a valid token. For example, a naive token concatenation macro might look like:

```
CAT(x, y) x##y
```

which would work fine in cases like `int i = CAT(1,2);` and expand to `int i = 12;` without issue. The problem comes about when the macro is invoked recursively, such as:

```
int i = CAT(CAT(1, 2), 3);
```

which will be greeted with:

```
warning 2446: pasting formed ')3', an invalid preprocessing token
int i = CAT(CAT(1, 2),3);
        ^
supplemental 893: expanded from macro 'CAT'
#define CAT(x,y) x##y
                  ^
```

followed by other parsing errors. One way to handle this is to use two macros:

```
#define CATX(x, y) x##y
#define CAT(x, y) XCAT(x, y)
```

2447 warning **'main' function should not be declared as '*string*'**
This message is issued when the `main` function is declared as `static`, `inline`, `constexpr`, or `deleted`. The C++ Standard forbids the `main` function to be declared with these specifiers.

2448 warning **'main' function should return type 'int'**
According to the C Standard, the `main` function must return `int` in a hosted environment but a return type other than `int` was specified for `main`. If you are targeting a freestanding/embedded platform or making use of non-standard extensions, you should suppress this message.

2450 warning **null character ignored**
A literal null character was encountered within the module being processed and will be ignored by PC-lint Plus.

2452 warning ***string* converts between pointers to integer types with different sign**
A pointer to an signed integer type was implicitly converted to or from a pointer to the corresponding unsigned integer type. For example:

```
void foo(int *p) {
    unsigned *up = p;   // Warning 2452
}
```

2453 **incompatible pointer to integer conversion *string string***
warning A pointer type was implicitly converted to an incompatible integer type. For example:

```
void foo(float *p) {
    int i = p;  // Warning 2453
}
```

2454 **incompatible pointer types *string string***
warning A pointer type was implicitly converted to an incompatible pointer type. For example:

```
void foo(float *pf) {
    int *pi = pf;  // Warning 2454
}
```

2455 **incompatible function pointer types *string string***
warning A function pointer type was implicitly converted to an incompatible function pointer type. For example:

```
void foo(int i) {
    int (*pf)(float) = &foo;    // Warning 2455
}
```

2456 **C++ language linkage specification encountered in C mode**
warning A C++ language linkage specification was encountered in a C module. For example:

```
extern "C++" int i;
```

This may indicate that a C++ module is incorrectly being processed in C mode or that a region of code that is only intended to be processed in C++ is not properly guarded (e.g. with `#ifdef __cplusplus`). Language linkage specifications in C mode are supported by some embedded compilers. If your compiler supports this, feel free to suppress this message.

2465 **redefinition of tag *type* will not be visible outside of this function**
warning A tag that was previously defined is being redefined in a function parameter list. While this is legal, it is suspect as this redefinition will only be visible within the function. It would be better to use another name and/or place the desired definition outside the function if the intention is to make the tag visible elsewhere.

2466 ***symbol* was used despite being marked as 'unused'**
warning The specified symbol was used despite being marked as unused, either via `#pragma` unused, the GCC `__attribute__` syntax, or with a C++11-style attribute specified. For example:

```
int i = 1;
#pragma unused(i)
int j [[gnu::unused]] = 2;
int k __attribute__((unused)) = 3;
```

Message `2466` will be issued if `i`, `j`, or `k` are subsequently used.

2491 **unknown expression '*string*' in sizeof will evaluate to 0, use -pp_sizeof to change the value used for evaluation**
warning

A `sizeof` expression was encountered inside of a preprocessor conditional. Furthermore, the expression appearing within `sizeof` was not previously registered with the `-pp_sizeof` option and will evaluate to zero. See the `-pp_sizeof` option for more information.

2501 **negation of value of unsigned type *type* yields a value of signed type *type* due to integral promotion**
warning

An unsigned integer type was promoted to a signed type as an operand to the unary minus operator. This may surprise those who are otherwise familiar with the common adage that applying unary minus to an unsigned type does not yield a negative value (see message `501`). For example: (assuming 16-bit shorts and 32-bit ints)

```
-(unsigned)5; // 2^32 - 5, type is still unsigned int
-(unsigned short)5; // -5, type is signed int
```

2586 ***string name* is deprecated**
warning

This message is issued when a entity is encountered that has been deprecated using either the C++14 deprecated attribute or the GCC deprecated attribute syntax. The type and name of the deprecated entity are provided in the message. If the deprecation contains a reason text, this is included as an additional string parameter as the end of the message. An `891` message provides the location of the actual deprecation. For example:

```
[[deprecated]] void foo();

void bar() {
    foo();
}
```

The use of `foo` on line 4 results in the message:

```
warning 2586: Function 'foo' is deprecated
    foo();
    ^
supplemental 891: Function 'foo' was marked deprecated here
        [[deprecated]] void foo();
                            ^
```

This message is not used to report the use of entities that are deprecated with the `-deprecate` option, such instances are instead reported by message `586`.

2623 **possible domain error for function *symbol*, argument *integer* (value=*string*) outside of accepted range (*string*)**
warning

Value tracking inferencing has determined that the value provided to a mathematical function is within a range that contains values that are not appropriate for the function and may result in a domain error. For example:

```
double foo(unsigned i) {
    if (i <= 10)
        acos(i);
}
```

will solicit the message:

```
warning 2623: possible domain error for function 'acos(double)', argument 1
   (value=0:10) outside of accepted range (between -1 and 1)
       acos(i);
            ^
```

as the valid range for the argument to `acos` is `[-1, 1]` and all that is known about the value provided is that it is between 0 and 10. To eliminate the diagnostic, the test should be corrected as in

```
if (i <= 1)
```

Supports MISRA C 2012 Directive 4.11 (Req)

2641 **implicit conversion of enum *symbol* to floating point type *type***
warning
An enumeration type was implicitly converted to a floating point type. Since enumerations are always represented using integral underlying types, it is suspicious to use an enumeration value in a floating point context. This message can be suppressed by using a cast.

2650 **constant '*integer*' out of range for '*string*' portion of compound comparison operator '*string*'**
warning
This message is issued when only the "greater than" or "less than" part of a "greater than or equal" or "less than or equal" compound comparison operator is out of range. For example (assuming 8-bit bytes):

```
1 void foo(unsigned char a) {
2     if (a == 255) { }   // Okay - 'a' could be equal to 255
3     if (a > 255) { }    // 650  - 'a' can't be greater than 255
4     if (a >= 255) { }   // 2650 - 'a' could be equal but not greater than 255
```

Message 2650 is issued on line 4 because while `a` could be equal to 255, it cannot be greater than 255 so the use of the `>=` operator is suspicious (perhaps `a` was intended to be compared to a different value). See also message 650 which is issued when the provided constant is out of range for the entire comparison operator.

2662 **pointer arithmetic on pointer that may not refer to an array**
warning
This message is issued instead of 662 when a pointer that appears likely not to refer to an array is subject to integer arithmetic. Addition, subtraction, and array subscripting are considered. Referring to the value itself with the operand zero is ignored. For example:

```
void f(int a) {
    int* p = &a;
    p[0] = 0;
    p[1] = 0;   // Warning 2662
    p + 0;
    p + 1;      // Warning 2662
}
```

2701 ***variable/function symbol* declared outside of header is not defined in the same source file**
info
The specified *symbol* was declared inside of a module but not defined inside the same module. If the *symbol* is defined in another module, it would be better to place the declaration of the *symbol* in a header and include that header in the modules that use the *symbol*.

2702 **static symbol *symbol* declared in header not referenced**
info The named static symbol was declared in a header included by the module but was not used within the including module. If the symbol had been declared in the module itself, warning 528 would be issued instead.

2703 **dangling else, add braces to body of parent statement to make intent explicit**
info A dangling `else` occurs when an `if/else` construct appears as the unbraced body of an `if` statement. In such cases, it may not be clear which of the `if` statements the `else` is intended to be associated with. For example:

```
int foo(int a, int b) {
    if (a)
    if (b)
        return 1;
    else
        return 0;
    return 2;
}
```

Is the `else` statement part of the `if (a)` or the `if (b)`? In C and C++, the `else` is associated with the closest preceding `if` that it is legal to be associated with so the `else` in the example is associated with `if (b)`. The message can be addressed by placing braces around the parent `if (a)` statement to make the intention explicit:

```
int foo(int a, int b) {
    if (a) {
        if (b)
            return 1;
        else
            return 0;
    }
    return 2;
}
```

2704 **potentially negating the most negative number**
info An integer value with the potential to equal the most negative possible integer was negated. In a two's complement representation, there is no positive equivalent to the most negative representable integer. For example:

```
void f(int a) {
    if (a < 0) {
        a = -a;
    }
    // Not safe to assume a is non-negative, negation of -2147483648
    // yields the same negative value in many compilers.
}
```

2705 **type qualifier(s) '*string*' applied to return type *has/have* no effect**
info A type qualifier was provided for the return type of a function but has no effect. Was the intention to qualify the function, a pointee type, a reference to the type returned, or something else? For example:

```
const int foo();
```

will be met with this message as `const` qualifier has no effect in this context.

2706 info **integer constant value does not match any enumerator in enumeration *type***
An integer constant is being used to assign a value to an enumeration type but the constant value does not match the value of any of the enumeration's enumerators. For example:

```
enum color { RED, GREEN, BLUE };

void foo(enum color);
void bar() {
    enum color c1 = RED;    // Okay
    enum color c2 = 0;      // Okay
    enum color c3 = 3;      // 2706
    foo(4);                 // 2706
}
```

The values 3 and 4 are not part of the enumeration '`color`' so `2706` will be issued in these cases.

2707 info **function *symbol* could be declared as 'noreturn'**
The specified function has no means of returning to its caller but this information is not included in the functions declaration via either the C11 `_Noreturn` keyword, the C++11 `noreturn` attribute, or the GCC `noreturn` attribute. Adding this information to the declaration may help clarify the purpose of the function.

2709 info **array subscript is of type 'char'**
A value of type '`char`' was used as the subscript to an array. '`char`' is a signed type on some platforms, relying on the signedness of '`char`' in this was is not wise.

2712 info **large pass-by-value parameter *symbol* of type *type* (*integer* bytes) for function *symbol***
The specified function was declared as taking a large object type by-value. It may be more efficient to have the function receive a pointer or reference instead. The threshold for determining what constitutes a large object is specified using the `-size` option.

2713 info **large return type *type* (*integer* bytes) for function *symbol***
A large object type is being returned by-value from the specified function; you might want to consider returning the object by pointer or reference instead. The threshold for determining what constitutes a large object is specified using the `-size` option.

2715 info **token pasting of ',' and __VA_ARGS__ is a GNU extension**
The token pasting operator `##` appeared between a comma and the `__VA_ARGS__` macro. While supported by several compilers as a mechanism by which to elide a trailing comma in a variadic macro, such a construct is technically undefined and could result in different behavior on a compiler that doesn't support this extension. See the discussion for the `frc` macro for more details.

2716 info **tentative array definition for variable *symbol* assumed to have one element**
This message is issued when a declaration for a variable of array type that acts as a tentative definition is encountered without a declared array size. A *tentative definition* in C is a file-scope declaration without an initializer that does not contain an `extern` storage class specifier. If the translation unit contains no external definition for an identifier, the C Standard specifies that it is defined with the composite type of the tentative

definition(s) for that identifier. In the case of an array without a size, this becomes an array with one element. This might represent an oversight in the program. If this was intentional, it would be clearer to define the array explicitly with one element.

2865 ***string***
info
A `#pragma message` directive was encountered. The text of this message is the string provided in the pragma.

2901 **stack usage information: *detail***
note
When generating stack reporting data, if the `fun` flag is set, this message will be issued once for each function in the stack report with *detail* containing the stack information for the function. See `-stack` for details.

2902 **discarded instance of post-*string* operator**
note
A post-increment or post-decrement operator was applied to a scalar in a context where the value will not be used (i.e. as an expression statement, as the left operand to the comma operator, or as the third clause of a `for` statement). A modern optimizing compiler is virtually guaranteed to elide the wasteful copy implied by such an operation when the value is not used but some may prefer to use pre-increment or pre-decrement operators instead for clarity and consistency. See also 1757 for potentially more serious cases involving user-defined types.

2932 **macro *name* used as header guard is followed by a #define of a different macro '*name*'**
note
A macro with a name that is different from the header guard macro was defined immediately after the header guard. It is conventional practice to define the macro used in the header guard check but this is not always done for every file; this message can be used to identify those that do not follow this pattern. The more egregious violations (those that define a macro with a very similar name) are flagged by message 2432.

15.5 Messages 3000-3999

3401 **parameter to move constructor *symbol* is an rvalue reference to const**
warning
There are relatively few valid reasons to declare a move constructor taking an rvalue reference to `const` but this construct could easily be formed by accident due to its similarity to a canonical copy constructor. This message will not be given if the move constructor is deleted, as this is occasionally useful.

3402 **lambda capture default captures 'this' by value**
warning
A lambda with reference capture still implicitly captures the `this` pointer by value if any non-static members are used. For clarity, some prefer to explicitly specify this in the capture list.

3403 **use of std::move on value of forwarding reference type *type*; was std::forward<*string*> intended?**
warning
A forwarding reference (sometimes referred to as a universal reference) was given as an argument to `std::move`. Either the formation of a forwarding reference instead of an rvalue reference or the use of `std::move` instead of `std::forward` was likely accidental. For example:

```
template<typename T>
void f(T&& t) {
    g(std::move(t)); // might unexpectedly move from the caller's lvalue
}
```

3405 ***symbol* is specified with C linkage but returns type *type* which is incompatible with C**
warning A function was specified as having C language linkage but the function returns a type that is not compatible with C so what is the point in having C linkage?

3406 **incomplete return type *type* for function *symbol* specified with C linkage**
warning A function that was specified as having C language linkage has an incomplete return type.

3407 ***symbol* should not return null unless declared with 'throw()' or 'noexcept'**
warning The implementation for an `operator new` function has the possibility to return null but the function is not declared with '`throw()`' or '`noexcept`'. `Operator new` functions should never return null except in these cases.

3408 **address of reference in comparison is never null in well-formed C++**
warning The address of a reference was directly compared to null. This is suspicious because the address of a reference can never be null in well-formed code

```
void foo(int &i) {
  int &ri = i;
  if (&i == 0) return;   // 3408
  if (&i != 0) return;   // 3408
}
```

3409 **address of reference in boolean context is never null in well-formed C++**
warning The address of a reference was used in a boolean context. This is suspicious because such an address can never be false. For example:

```
void foo(int &i) {
  int &ri = i;
  if (&i) return;    // 3409
  if (&ri) return;   // 3409
  if (!&i) return;   // 3409
}
```

3410 **conversion function converting *type* to itself will never be used**
warning A class contains a conversion function that converts to the *type* of the class itself. This is suspicious because such a conversion function will never be called. For example:

```
class X {
   operator X();
};
```

will elicit this message.

3411 **conversion function converting *type* to its base class *type* will never be used**
warning A class contains a conversion function that converts to the *type* of its base class. This is suspicious because such a conversion function will never be called. For example:

```
class X { };
class Y : public X {
```

```
    operator X();
};
```

will elicit this message.

3412 ***type* has virtual functions but non-virtual destructor**
warning
A class contains at least one virtual function but has a non-private, not-virtual destructor. Classes that may be used as base classes should always have virtual destructors to ensure that instances of derived classes that are deleted through a pointer to the base class are properly destructed. The declaration of a virtual function implies that this class is meant to be used as a base class and as such it should provide a virtual destructor.

3413 ***delete/destructor* called on non-final *type* that has virtual functions but non-virtual destructor**
warning
This message is similar to `3412` but while the former warns about the potential problem that could arise from having a non-virtual destructor, this message warns when delete is applied to such an object.

3414 ***delete/destructor* called on *type* that is abstract but has non-virtual destructor**
warning
This message is similar to `3413` but is reported for abstract types.

3415 **pointer initialized to temporary array**
warning
A pointer is being initialized with a temporary array, which will be destroyed at the end of the containing expression making it impossible to safely dereference the pointer before assigning a new value to it. For example:

```
struct S { int array[10]; };

void f() {
    int *pi = S().array;   // warning 3415, array pointed to by pi
                           // will cease to exist after initialization.
}
```

3416 **'this' pointer used in boolean context is never null**
warning
The `this` pointer was used in a boolean context such as:

```
if (this) ...
```

Was this a mistake? The `this` pointer is never null in well-formed C++ so such a test is suspect.

3417 **'this' pointer used in comparison is never null**
warning
The `this` pointer is explicitly tested for null such as

```
if (this == 0)
```

Was this a mistake? The `this` pointer is never null in well-formed C++ so such a test is suspect.

3418 **'reinterpret_cast' *string* class *symbol* *string* its *string* *symbol* behaves differently than 'static_cast'**
warning
A `reinterpret_cast` was used to cast between a class type and a base class type in an unsafe way; `static_cast` should probably be used instead.

3420 **extraneous template parameter list in template specialization**
warning
An extraneous template parameter list was provided in the declaration of a template specialization. For example:

```
template <typename T>
T foo(T);

template<>       // warning 3420
template<>
int foo(int);
```

3421 ***string* template partial specialization contains *string* that cannot be deduced so the specialization will never be used**
warning
In the partial specialization of a class or variable template, the presence of one or more template parameters that cannot be deduced means that the specialization will never be used. Was this a mistake?

3423 ***case value/enumerator value/non-type template argument/array size/constexpr if cannot be narrowed from type to type***
warning
A case value, enumerator value, non-type template argument, or array size was provided that cannot be narrowed to the required type. For example:

```
void foo(unsigned u) {
    switch(u) {
    case -1:         // warning 3423, cannot narrow -1 to unsigned
        break;
        ...
        }
}

template <unsigned char I>
struct S { unsigned char value = I; };
S<300> s;          // warning 3423, cannot narrow 300 to unsigned char
```

3424 **constexpr *function/constructor* never produces a constant expression**
warning
A function marked as `constexpr` must contain at least one code path that produces a constant expression to be used in a context where a constant expression is required. The specified function was marked as `constexpr` but does not ever produce a constant expression so the use of `constexpr` is suspect.

3425 **type *type* cannot be narrowed to *type* in initializer list**
warning
Inside an initializer list, a prohibited implicit narrowing conversion would be required to perform the initialization. The prohibited implicit conversion is one that is never allowed in list initialization. For example:

```
int i = { 3.0 };
```

will elicit this message because conversion from a floating point type to a integral type is required to perform the initialization but is not an allowed implicit conversion within an initializer list. The issue can be corrected by correcting the type used in the initializer, casting the type, or not using list initialization.

3426 **non-constant-expression cannot be narrowed from type *type* to *type* in initializer list**
warning
Inside an initializer list, a prohibited implicit narrowing conversion would be required to perform the

initialization. The implicit conversion is of a type that is allowed for constant expressions but not for the provided expression. For example:

```
extern int i;
float f = { i; };
```

will elicit this message while:

```
float f = { 3 };
```

will not. The issue can be corrected by correcting the type used in the initializer, casting the type, or not using list initialization.

3427 warning **constant expression evaluates to *string* which cannot be narrowed to type *type***

Inside an initializer list there is an implicit conversion of a constant expression to a type where the value cannot be represented exactly, which is prohibited. For example:

```
unsigned char c = { 1234 };
```

will elicit this message, assuming 8-bit `chars`. The message can be avoided by correcting the value, using a cast, or by not employing list initialization.

3428 warning **out-of-line declaration of a member must be a definition**

A `class` member that is declared out-of-line (outside of the `class` declaration) must have a definition. For example:

```
class A {
    void foo();
}

void A::foo();
```

The second declaration of `A::foo` must contain a definition.

3429 warning **parenthesized initialization of a member array is a GNU extension**

In C++11, an array member can be initialized in a member initialization list using extended initializer syntax, e.g.:

```
class A {
    A() : array { 0 } { };
    int array[10];
}
```

A deprecated GNU extension allowed an array member to be initialized using the syntax for initializing class types:

```
A() : array({ 0 });
```

This parenthesis around the initializer here are not Standard.

3430 warning **taking the address of a temporary object of type *type***

An attempt was made to take the address of a temporary object. For example:

```
struct X { ... };

void foo() {
```

```
    &X();       // warning 3430
}
```

3431 in-class initializer for static data member of type *type* requires 'constexpr' specifier

warning

A `const static` data member of integral type may be initialized in its in-class declaration with a constant expression. For non-integral types, the member must be declared with `constexpr`, e.g.:

```
struct A {
    const static int i = 3;
    constexpr static float f = 3.0;
};
```

This message is issued for non-integral static data members with an in-class initializer of a type for which `constexpr` is expected but not provided.

3432 invalid suffix on literal, C++ 11 requires a space between literal and identifier

warning

A literal appeared adjacent to an identifier but there was no space separating the two and the identifier was not a valid suffix for the literal.

3450 subtracting value of member *symbol* from the address referred to by the 'this' pointer; use of -> to access the member may have been intended

warning

The integral or pointer value of a member of `this`, accessed implicitly through the current object, was subtracted from the `this` pointer. This is almost certainly a mistake where the `>` in `->` was forgotten.
For example:

```
struct X {
    bool value;
    bool getValue() const {
        return this-value; // intended to be this->value
    }
};
```

If for some reason explicitly applying a negative offset to the `this` pointer based on a member value is actually desired then the member name can be enclosed in parentheses to avoid confusion.

3701 use of *symbol* implicitly invokes converting constructor *symbol*; *symbol* could be used

info

A `push`, `push_back`, or `insert` function is called in a situation where `emplace_back` or `emplace` could be used instead.

3702 lambda capture default captures 'this' by value

info

The `this` pointer was implicitly captured due to a member access inside a lambda. For clarity, some prefer to explicitly specify this in the capture list.

3703 ellipsis at this point creates a C-style varargs function

info

An ellipsis was encountered, which was probably intended to declare a function parameter pack but instead declares a variable argument function. For example:

```
template <typename... T>
void foo() {
    bar([] {
```

```
        void g(T t...);     // warning 3703, probably meant g(T... t);
    }...);
}
```

3704 **empty parentheses here declare a function, not a variable**
info
A set of empty parenthesis were added to what would otherwise be interpreted as a variable declaration but instead results in the declaration of a function. For example:

```
struct S {
    S(int a = 0) : _a(a) { }
    int _a;
};

void foo() {
    S s1(1);     // OK, declares and initializes variable s1
    S s2();      // warning 3704, declares a function s2
}
```

`s2` is interpreted as a function that returns type `S` and takes not arguments, not a zero-initialized variable as presumably intended.

There are multiple ways to force interpretation of a variable, e.g.:

```
S s3(0);    // OK
S s4 = S(); // OK, initialize via temporary
S s5{};     // OK, C++11 uniform initialization
```

See also message 3705.

3705 **parenthetic disambiguation results in function declaration**
info
A syntactic construct was encountered that could be interpreted as either a variable declaration or a function declaration (sometimes referred to as the "most vexing parse"). The C++ disambiguation rules require that it be interpreted as a function declaration, which may not be what the programmer intended. For example:

```
struct X { };
struct Y {
    Y(const X&);
};

void foo() {
    Y y(X());   // warning 3705
}
```

Here `y` is interpreted as a function that returns an object of type `Y` and takes a single parameter that is a pointer to a function taking no arguments and returning type `X`. In particular, it is *not* interpreted as a declaration of an object of type `Y` initialized with a temporary of type `X` as was almost certainly intended.

There are several ways to force interpretation of a variable declaration. In C++11 and later the simplest way is to employ uniform initialization syntax, for example any of the following would work:

```
Y y1(X{});    // OK, variable declaration
Y y2{X()};    // OK, variable declaration
Y y3{X{}};    // OK, variable declaration
```

Prior to C++11, an extra pair of parenthesis can be used to force the desired interpretation, e.g.:

```
Y y4((X()));   // OK, variable declaration
```

The same issue can appear with casts, for example:

```
void foo(double d) {
    int i( int(d) );    // warning 3705
}
```

In this case, `i` is not a variable initialized with the truncated value of `d` but rather a function returning `int` and taking `int`. In addition to the methods mentioned above to force a variable declaration, the functional cast can be converted to a C-style cast or a named cast, e.g.:

```
int i1( (int) d );              // OK, variable declaration
int i2( static_cast<int>(d) );  // OK, variable declaration
```

3706 **abstract class *symbol* marked '*final/sealed*'**
info An abstract class (one that contains at least one pure virtual specifier) was marked as `final` or `sealed` preventing the class from being used as a base class. Since abstract classes cannot be instantiated, what would be the purpose of having abstract class that cannot be inherited from?

3707 **unknown linkage language '*string*'**
info C++ defines the language linkage specifiers `"C"` and `"C++"`. Other specifiers may be supported by compilers as an extension with implementation-defined semantics. This message is issued when a language linkage specifier other than `"C"` or `"C++"` is encountered. For example:

```
extern "C" int a;       // Okay
extern "C++" int b;     // Okay
extern "ADA" int c;     // Info 3707
```

3901 **reference to *data member/member function symbol* of *symbol* does not use an explicit 'this->'**
note A non-static data member or function was referenced inside of the containing class with an implicit `this` object instead of using `this->`*`member`* to access the member. For example:

```
struct A {
    int value;
    int getValue() { return this->value; }  // OK, explicit this->
    void setValue(int v) { value = v; }     // note 3901
};
```

Some authors suggest always using `this->` to access members to prevent the potential for confusion when local objects or functions with the same name as a member exist in the same scope. See also message 578, which will be issued if a local symbol is declared that hides a member.

3902 **thrown object of type *type* is not a class derived from std::exception**
note This message is issued where a throw-expression initializes an exception object that is not derived from `std::exception`. The point is to have a type that can be caught by

```
catch(std::exception & p)
```

instead of

```
catch(...)
```

This way, in a situation where it's necessary to catch everything, information about the kind of error can at least be logged or translated.

15.6 Messages 4000-5999

Messages in this range are reserved for mapped compiler errors and detailed descriptions are not provided.

15.7 Messages 8000-8999

8000 ***string***
info Messages in the 8xxx range are reserved for user-defined diagnostics, see `+message` for more information.

15.8 Messages 9000-9999

9001 **octal constant '*string*' used**
note An octal constant appears in the code. Octal constants may be inadvertently interpreted by engineers as decimal values. This message is not issued for a constant zero written as a single digit.
Supports MISRA C 2004 Rule 7.1 (Req)
Supports MISRA C 2012 Rule 7.1 (Req)

9003 **could define global variable *symbol* within function *string***
note A variable was declared at global scope but only utilized within one function. Moving the declaration of this variable to that function reduces the chance the variable will be used incorrectly.
Supports MISRA C 2004 Rule 8.7 (Req)
Supports MISRA C 2012 Rule 8.9 (Adv)

9004 **object/function *symbol* previously declared**
note The named symbol was declared in multiple locations, not counting the point of definition for that symbol. Declaring a symbol in one location and in one file helps to ensure consistency between declaration and definition as well as avoiding the risk of conflicting definitions across modules.
Supports MISRA C 2004 Rule 8.8 (Req)
Supports MISRA C 2012 Rule 8.5 (Req)
Supports MISRA C++ Rule 3-2-3 (Req)

9005 **cast drops *detail* qualifier(s)**
note A cast attempted to remove the qualifiers from an object to which a pointer points or a reference refers. Doing so can result in undesired or unexpected modification of the object in question and may result in an exception being thrown.
Supports MISRA C 2004 Rule 11.5 (Req)
Supports MISRA C 2012 Rule 11.8 (Req)
Supports MISRA C++ Rule 5-2-5 (Req)
Supports MISRA C++ Rule 5-18-1 (Req)

9006 **'sizeof' used on expression with side effect**
note If the operand of the `sizeof` operator is an expression, it is not usually evaluated. Attempting to apply `sizeof` to such an expression can result, therefore, in code one expects to be evaluated actually not being evaluated and the side-effects not taking place. This message is not given if the operand is an lvalue of volatile qualified type and is not a variably-lengthed array.
Supports MISRA C 2004 Rule 12.3 (Req)
Supports MISRA C 2012 Rule 13.6 (Mand)
Supports MISRA C++ Rule 5-3-4 (Req)

9007 side effects on right hand of logical operator, '*string*'

note The right hand side of the `||` and `&&` operators is only evaluated if the left hand side evaluates to a certain value. Consequently, code that expects the right hand side to be evaluated regardless of the left hand side can produce unanticipated results.

Supports MISRA C 2004 Rule 12.4 (Req)
Supports MISRA C 2012 Rule 13.5 (Req)
Supports MISRA C++ Rule 5-14-1 (Req)

9008 comma operator used

note The comma operator is thought by some to reduce readability in code.

Supports MISRA C 2004 Rule 12.10 (Req)
Supports MISRA C 2012 Rule 12.3 (Adv)
Supports MISRA C++ Rule 5-18-1 (Req)

9009 possible use of floating point loop counter

note The use of floating point variables as loop counters can produce surprising behavior if the accumulation of rounding errors results in a different number of iterations than anticipated.

Supports MISRA C 2004 Rule 13.4 (Req)
Supports MISRA C 2012 Rule 14.1 (Req)

9011 multiple loop exits

note More than one `break` statement or `goto` statement is used to terminate a loop. Minimizing the number of exits from a loop is thought by some to reduce visual complexity of the code.

Supports MISRA C 2004 Rule 14.6 (Req)
Supports MISRA C 2012 Rule 15.4 (Adv)
Supports MISRA C++ Rule 6-6-4 (Req)

9012 body should be a compound statement

note Multiple authors have advised making sure the body of every *iteration-statement* and *selection-statement* be a *compound-statement*. However, no `{` was seen to begin the *compound-statement*.

Supports MISRA C 2004 Rule 14.8 (Req)
Supports MISRA C 2004 Rule 14.9 (Req)
Supports MISRA C 2012 Rule 15.6 (Req)
Supports MISRA C++ Rule 6-3-1 (Req)
Supports MISRA C++ Rule 6-4-1 (Req)

9013 no 'else' at end of 'if ... else if' chain

note An `if...else if` chain was seen without a final `else` statement. Providing such a statement helps to act as an analog to the `default` case of a `switch-statement`.

Supports MISRA C 2004 Rule 14.10 (Req)
Supports MISRA C 2012 Rule 15.7 (Req)
Supports MISRA C++ Rule 6-4-2 (Req)

9014 switch without default

note A `switch-statement` was found without a `default` case. Providing such a case provides defensive programming.

Supports MISRA C 2004 Rule 15.3 (Req)

Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C 2012 Rule 16.4 (Req)

9015 note **macro '*name*' appearing in argument *integer* of macro '*name*' is used both with and without '#/##' and is subject to further replacement**

In the expansion of a function-like macro, a macro argument was used both as an operand to the stringizing or pasting operators and was also used in a way in which it was subject to further macro replacement. For example:

```
#define M1 123
#define FM(x) ident_ ## x + x
...
FM(10);     // Okay, 10 is not a macro
FM(M1);     // 9015, M1 both expanded and used with ##
```

The `FM` macro uses the paramater `x` as an operand to the token pasting operator (where a macro argument would not be expanded) and in a context where a macro argument would be expanded. This example expands to:

```
ident_10 + 10;
ident_M1 + 123;
```

In the second invocation, part of the expansion contains the unexpanded macro and another contains the result of the expanded macro argument. This may be confusing and lead to unexpected results.
Supports MISRA C 2012 Rule 20.12 (Req)

9016 note **performing pointer arithmetic via *addition/subtraction***

Array indexing is thought, by some, to be more readily understood and less error prone than other forms of pointer arithmetic.
Supports MISRA C 2004 Rule 17.4 (Req)
Supports MISRA C 2012 Rule 18.4 (Adv)
Supports MISRA C++ Rule 5-0-15 (Req)

9017 note ***incrementing/decrementing* pointer**

While at least one standards organization cautions against using any pointer arithmetic besides array indexing, the use of increment or decrement operators with pointers may represent an intuitive application and illustration of the underlying logic. Consequently, such constructs are separated from message 9016 and placed under this one, allowing a more fine tuning of Lint diagnostics.
Supports MISRA C 2004 Rule 17.4 (Req)

9018 note **union *symbol* declared**

Depending upon padding, alignment, and endianness of union, as well as the size and bit-order of their members, the use of unions can result in unspecified, undefined, or implementation defined behavior, prompting some to advise against their use.
Supports MISRA C 2004 Rule 18.4 (Req)
Supports MISRA C 2012 Rule 19.2 (Adv)
Supports MISRA C++ Rule 9-5-1 (Req)

9019 note **declaration of *symbol* before #include**

The symbol mentioned in *string* was seen in a module with a subsequent `#include` directive. It can be argued

that collecting all `#include` directives at the beginning of the module helps improve code readability and helps reduce risk of undefined behavior resulting from any use of the ISO standard library before the relevant `#include` directive.
Supports MISRA C 2004 Rule 19.1 (Adv)
Supports MISRA C 2012 Rule 20.1 (Adv)
Supports MISRA C++ Rule 16-0-1 (Req)

9020 **header file name with non-standard character '*detail*'**
note The use of non-standard characters in `#include` directives results in undefined behavior.
Supports MISRA C 2004 Rule 19.2 (Adv)
Supports MISRA C 2012 Rule 1.3 (Req)
Supports MISRA C 2012 Rule 20.2 (Req)
Supports MISRA C++ Rule 16-2-4 (Req)
Supports MISRA C++ Rule 16-2-5 (Adv)

9021 **use of '#undef' is discouraged: '*detail*'**
note The use of the `#undef` directive can lead to confusion about whether or not a particular macro exists at a randomly given point of code.
Supports MISRA C 2004 Rule 19.6 (Req)
Supports MISRA C 2012 Rule 20.5 (Adv)
Supports MISRA C++ Rule 16-0-3 (Req)

9022 **unparenthesized macro parameter '*string*' in definition of macro '*string*'**
note Multiple authors have cautioned against the use of unparenthesized macro parameters in cases where the parameter is used as an expression. If care is not taken, unparenthesized macro parameters can result in operator precedence rules producing expressions other than intended.
Supports MISRA C 2004 Rule 19.10 (Req)
Supports MISRA C++ Rule 16-0-6 (Req)

9023 **multiple use of stringize/pasting operators in definition of macro *name***
note Multiple use of such operators is thought by some to increase risk of undefined behavior.
Supports MISRA C 2004 Rule 19.12 (Req)
Supports MISRA C 2012 Rule 1.3 (Req)
Supports MISRA C++ Rule 16-3-1 (Req)

9024 ***pasting/stringize* operator used in definition of *object-like/function-like* macro '*string*'**
note The use of token pasting (`##`) and stringizing (`#`) preprocessor operators is thought by some to reduce code clarity and increase the risk of undefined behavior.
Supports MISRA C 2004 Rule 19.13 (Adv)
Supports MISRA C 2012 Rule 20.10 (Adv)
Supports MISRA C++ Rule 16-3-2 (Adv)

9025 **more than two levels of pointer indirection**
note Three or more levels of pointer indirection may make it harder to understand the code.
Supports MISRA C 2004 Rule 17.5 (Adv)
Supports MISRA C 2012 Rule 18.5 (Adv)
Supports MISRA C++ Rule 5-0-19 (Req)

9026 function-like macro, '*macro*', defined
note Multiple authors have expressed reasons why a function, when possible, should be used in place of a function-like macro.
Supports MISRA C 2004 Rule 19.7 (Adv)
Supports MISRA C 2012 Directive 4.9 (Adv)
Supports MISRA C++ Rule 16-0-4 (Req)

9027 *essential-type* value is not an appropriate *string* operand to *operator*
note Out of concern for unspecified, undefined, and/or implementation defined behavior, some standards urge restrictions on certain types of operands when used with certain operators.
Supports MISRA C 2012 Rule 10.1 (Req)

9028 *essential-type* value is not an appropriate *string* operand to *operator*
note MISRA C 2012 has defined the concept of *essentially character type* and placed restrictions on the use of expressions with such a type.
Supports MISRA C 2012 Rule 10.2 (Req)

9029 *essential-type* value and *essential-type* value cannot be used together as operands to *operator*
note MISRA C 2012 has defined the concept of *essentially type* and placed restrictions on the use of expressions with certain types with respect to binary operators.
Supports MISRA C 2012 Rule 10.4 (Req)

9030 cannot cast *essential-type* value to *essential-type* type
note MISRA C 2012 has defined the concept of *essential type* and placed restrictions on the use of casts between certain types.
Supports MISRA C 2012 Rule 10.5 (Adv)

9031 cannot assign a composite expression of type '*essential-type*' to an object of wider type '*essential-type*'
note MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on assignments of the former.
Supports MISRA C 2012 Rule 10.6 (Req)

9032 *left/right* operand to *operator* is a composite expression of type '*essential-type*' which is smaller than the *left/right* operand of type '*essential-type*'
note MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on operands to binary operators when at least one of the operands meets the definition of the former concept.
Supports MISRA C 2012 Rule 10.7 (Req)

9033 cannot cast '*essential-type*' to *wider/different* essential type '*essential-type*'
note MISRA C 2012 has defined the concepts of *composite expression* and *essential type* and placed restrictions on casts of the former. This message, when given, is also followed by text explaining why the cast is considered "impermissible".
Supports MISRA C 2012 Rule 10.8 (Req)

9034 **cannot assign '*essential-type*' to *narrow/different* essential type '*essential-type*'**

note MISRA C 2012 has defined the concept of *essential type* and placed restrictions on assignments in relation to such types.

Supports MISRA C 2012 Rule 10.3 (Req)

9035 **variable length array *symbol* of type *type* declared**

note Variable length arrays can introduce unspecified behavior and runtime-dependent undefined behavior. As of C11 it is not required that implementations support this feature. For these reasons, the use of VLAs is often discouraged.

Supports MISRA C 2012 Rule 18.8 (Req)

9036 **essential type of condition is '*essential-type*' but should be boolean**

note MISRA C 2012 has defined the concept of *essentially Boolean* type and requires the conditional expressions of all `if` and `iteration-statements` comply with this definition.

Supports MISRA C 2012 Rule 14.4 (Req)

9037 **conditional of *#if/#elif* does not evaluate to 0 or 1**

note Some urge such a practice in the interest of strong typing.

Supports MISRA C 2012 Rule 20.8 (Req)

9038 **flexible array member declared**

note Flexible array members can alter the behavior of `sizeof` in surprising ways. Additionally, flexible array members often require dynamic memory allocation, which may be problematic in safety critical code.

Supports MISRA C 2012 Rule 18.7 (Req)

9039 **potentially confusing *hexadecimal/octal* escape sequence usage**

note An octal or hexadecimal escape sequence has been detected within a string or character literal that is not immediately followed by another escape sequence or end of literal.

Supports MISRA C 2012 Rule 4.1 (Req)

9041 **goto appears in block *string* which is not nested in block *string* which contains label *symbol***

note It has been deemed safer by some experts that the block (i.e., compound statement) containing the `goto` should be the same as or nested within the block containing the label. Thus

```
{ label: { goto label; } }
```

is permitted but

```
{  goto label;  {  label: ; }  }
```

is not. To assist the programmer, the message refers in the blocks using an identification code (e.g. "1.2.1"). This identification scheme is defined as follows:

1. The outer block has an identification of 1.
2. If a particular block is identified by `x` then its immediate subblocks, if any, are identified as `x.1`, `x.2`, `x.3`, etc.

Thus in the following 'code',

```
{ { } {{label: } { } } }
```

`label` lies in block 1.2.1.
Supports MISRA C 2012 Rule 15.3 (Req)
Supports MISRA C++ Rule 6-6-1 (Req)

9042 **departure from MISRA switch syntax: *detail***
note A *switch-statement* was found that does not comply with the MISRA *switch-statement* syntax. *detail* contains a description of the departure.
Supports MISRA C 2004 Rule 15.0 (Req)
Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C++ Rule 6-4-3 (Req)

9043 **static keyword between brackets of array declaration**
note Some advocate against using the keyword `static` in array declarations due to a perceived increased risk of undefined behavior.
Supports MISRA C 2012 Rule 17.6 (Mand)

9044 **function parameter *symbol* modified**
note It has been advocated that function parameters be first copied to local variables where they can be modified rather than modifying the parameters directly.
Supports MISRA C 2012 Rule 17.8 (Adv)

9045 **complete definition of *symbol* is unnecessary in this translation unit**
note Some advise against including structure definitions unless the definition is required for the current module.
Supports MISRA C 2012 Dir 4.8 (Adv)

9046 ***symbol* is typographically ambiguous with respect to '*string*' when *detail***
note Some have warned against the use of identifiers that may be considered typographically ambiguous. In addition to the name of the previously seen symbol, the reasons Lint considers the identifiers to be ambiguous and the location of said previous symbol are provided in the message, if available.
Supports MISRA C 2012 Dir 4.5 (Adv)
Supports MISRA C++ Rule 2-10-1 (Req)

9047 **FILE pointer dereferenced**
note At least one standards organization urges against this practice, directly or indirectly.
Supports MISRA C 2012 Rule 22.5 (Mand)

9048 **unsigned integer literal without a 'U' suffix**
note An integer literal of unsigned type was found without a 'U' suffix.
Supports MISRA C 2004 Rule 10.6 (Req)
Supports MISRA C 2012 Rule 7.2 (Req)

9049 **increment/decrement operation combined with other operation with side-effects**
note An expression was seen involving an increment or decrement operator and the expression also contained potential side-effects other than those resulting from said operator. For the purpose of this message, a function call is always considered to have potential side-effects.
Supports MISRA C 2004 Rule 12.13 (Adv)

Supports MISRA C 2012 Rule 13.3 (Adv)
Supports MISRA C++ Rule 5-2-10 (Adv)

9050 dependence placed on operator precedence (operators '*operator*' and '*operator*')
note Reliance on operator precedence was found in a particular expression. Using parentheses, it is felt, helps clarify the order of evaluation.
Supports MISRA C 2004 Rule 12.1 (Adv)
Supports MISRA C 2012 Rule 12.1 (Adv)

9051 macro '*string*' defined with the same name as a C keyword
note A macro was defined with the same name as an ISO C keyword. The use of such a macro causes undefined behavior.
Supports MISRA C 2012 Rule 20.4 (Req)

9052 macro '*string*' defined with the same name as a C++ keyword
note A macro was defined with the same name as an ISO C++ keyword. The use of such a macro causes undefined behavior.
Supports MISRA C++ Rule 17-0-1 (Req)

9053 the shift value is at least the precision of the essential type of the left hand side
note MISRA 2012 defines the notion of an "essential type". A quantity with a certain essential type, as defined by MISRA, was left shifted by a number exceeding the number of bits used to represent that essential type.
Supports MISRA C 2012 Rule 12.2 (Req)

9054 designated initializer used with array of unspecified dimension
note It has been advocated, when arrays initializers contain designators, the dimension of the array should be explicitly stated in the declaration. The initializer of the array in question has been found in violation of this recommendation.
Supports MISRA C 2012 Rule 9.5 (Req)

9055 most closely enclosing compound statement of this '*string*' label is not a switch statement
note Labels nested inside of compound statements within the corresponding `switch` are legal but can reduce comprehension and lead to unstructured code.
Supports MISRA C 2004 Rule 15.1 (Req)
Supports MISRA C 2012 Rule 16.2 (Req)
Supports MISRA C++ Rule 6-4-4 (Req)

9056 inline function *symbol* defined with storage-class specifier *string*
note This message is issued for all inline functions defined with a storage-class specifier. `+estring` can be used to find all inline functions defined with a specific specifier. For example, `+estring(9056, extern)` will report all inline functions defined with extern.
Supports MISRA C 2012 Rule 8.10 (Req)

9057 lowercase L follows 'u' in literal suffix
note A lowercase letter "`l`" is used inside of a literal suffix following an upper or lowercase letter `u`. With some fonts, the lowercase letter "`l`" can be easily confused with the number one. This is less likely to happen when

there is a "u" between the number and the "l" (as in `35ul`), but some coding standards forbid the use of "l" in any literals. Message 620 reports the more suspicious case where the "l" immediately follows a number (as in `35l`).
Supports MISRA C 2012 Rule 7.3 (Req)

9058 note **tag *symbol* unused outside of typedefs**
A tag was used only in the course of creating a `typedef`. Was the tag unused by mistake (say a recursive reference inside the body of the struct was accidentally omitted)? Such tags are most often redundant and can be eliminated.
Supports MISRA C 2012 Rule 2.4 (Adv)

9059 note **C comment contains C++ comment**
A C++-style comment was seen inside a C-style comment. This can be confusing.
Supports MISRA C 2012 Rule 3.1 (Req)

9060 note **trigraph in comment**
A trigraph was seen inside a comment. Since trigraphs are translated before preprocessing, a trigraph sequence like `??/` can have surprising results, especially in a C++ style comment where the trigraph sequence translates into a backslash.
Supports MISRA C 2012 Rule 4.2 (Adv)

9063 note **no comment or action in else**
An `else`-branch was seen that contained neither a comment nor an actionable statement. At least one standards organization cautions against such "empty else" branches.
Supports MISRA C 2004 Rule 14.10 (Req)
Supports MISRA C 2012 Rule 15.7 (Req)

9064 note **goto references earlier label *symbol***
A `goto` makes reference to a label appearing earlier in the code. At least one author recommends all such statements reference points later in the code in an attempt to reduce visual code complexity.
Supports MISRA C 2012 Rule 15.2 (Req)
Supports MISRA C++ Rule 6-6-2 (Req)

9066 note **C++ comment contains C comment**
A C-style comment was seen inside a C++-style comment. This can result in confusion.
Supports MISRA C 2012 Rule 3.1 (Req)

9067 note **extern array declared without size or initializer**
An array was declared without a dimension. At least one standards organization advises against such a practice in the interest of safety. Note this message is not given if the array is initialized at the time of declaration.
Supports MISRA C 2004 Rule 8.12 (Req)
Supports MISRA C 2012 Rule 8.11 (Adv)
Supports MISRA C++ Rule 3-1-3 (Req)

9068 note **partial array initialization**

An array has been initialized only partly. Providing an explicit initialization for each element of an array makes it clear every element has been considered. This diagnostic is not issued if the array is initialized with a {0} initializer or if the initializer consists entirely of designated initializers or if the array is initialized using a string literal. See also 785.
Supports MISRA C 2004 Rule 9.2 (Req)
Supports MISRA C 2012 Rule 9.3 (Req)

9069 note **in initializer for symbol *symbol*, initializer of type *type* needs braces or designator**
In the initializer for a variable declared with aggregate (array or structure) or union type, there were insufficient braces or designators necessary to make clear which members/elements are initialized to which values. More specifically, the initializer is expected to be fully braced, e.g. with braces appearing at the beginning of every aggregate sub-object being explicitly initialized, with the following exceptions:

- If all of the initializers for a particular sub-object are designated initializers, braces are not required for that sub-object.
- String literals may be used to initialize arrays.
- An aggregate sub-object may be initialized with an object of compatible type.
- The idiomatic { 0 } may be used to initialize sub-objects to an arbitrary depth without providing nested braces.

For example:

```
enum wk_type { FIRE, ICE };

struct monster {
    const char name[10];
    int hp;
    struct weakness {
        enum wk_type wk;
        double dmg_mult;
    } weak[2];
};

// Okay - all initialized sub-objects are braced, array 'name' initialized with string literal
struct monster goblin1 = {"goblin", 10, {{ICE, 2.0}, {FIRE, 1.5}}};

// 9069 - the second element of the 'weak' array is not braced
struct monster goblin2 = {"goblin", 10, {{ICE, 2.0}, FIRE, 1.5}};

// Okay - only initialized part of non-braced sub-object uses designated initializer
struct monster goblin3 = {"goblin", 20, .weak[0].wk = FIRE};

// 9069 - '1' initializes part of sub-object that is not braced
struct monster goblin4 = {"goblin", 10, .weak[0].wk = FIRE, 1};

// 9069 - initialized sub-object 'struct weakness [0]' needs additional braces
struct monster goblin5 = {"goblin", 40, {1}};

// Okay - exception for sub-objects initialized with { 0 }
struct monster goblin6 = {"goblin", 40, {0}};
```

Supports MISRA C 2012 Rule 9.2 (Req)

9070 function '*name*' is recursive
note
The named function has been found to potentially call itself, either directly or indirectly. Recursion carries with it the danger of exceeding available stack space, which can lead to a run-time failure. All else being equal, the more that recursion is constrained, the easier determining the worst-case stack usage can be.
Supports MISRA C 2004 Rule 16.2 (Req)
Supports MISRA C 2012 Rule 17.2 (Req)
Supports MISRA C++ Rule 7-5-4 (Adv)

9071 defined macro '*name*' is reserved to the compiler
note
A macro was defined that is reserved to the compiler. Such definition results in undefined behavior.
Supports MISRA C 2004 Rule 20.1 (Req)
Supports MISRA C 2012 Rule 21.1 (Req)
Supports MISRA C++ Rule 17-0-1 (Req)

9072 parameter *integer* of function *symbol* has different name than previous declaration (*symbol* vs *symbol*)
note
The parameter of function *symbol* specified by *integer* has a parameter name that differs from the name of a previous declaration of the same function. Using inconsistent names within declarations of the same function can be confusing and result in misuse.
Supports MISRA C 2004 Rule 16.4 (Req)
Supports MISRA C 2012 Rule 8.3 (Req)
Supports MISRA C++ Rule 8-4-2 (Req)

9073 parameter *integer* of function *symbol* has type alias name type difference with previous declaration (*type* vs *type*)
note
In a function declaration or definition, the specified parameter is declared with a *type* that, while technically identical, uses a different name for the *type* than was used for the parameter in a previous declaration. For example:

```
typedef int INT;
void foo(int i);
void foo(INT i) {
    ...
}
```

would yield this message as the parameter `i` in function `foo` is declared as an int in both cases but in the definition the `typedef` name `INT` is used while in the preceding declaration the name `INT` is not employed. Such inconsistencies can result in unnecessary confusion.
Supports MISRA C 2004 Rule 8.3 (Req)
Supports MISRA C 2012 Rule 8.3 (Req)
Supports MISRA C++ Rule 3-9-1 (Req)

9074 conversion between pointer to function type *type* and differing type *type*
note
A conversion was seen between a pointer to function and a different type. The conversion of a pointer to a function into or from a pointer to object, pointer to incomplete type, or pointer to void results in undefined behavior and, consequently, at least one standards organization advises against such practice. This diagnostic is suppressed if the conversion is to void.
Supports MISRA C 2012 Rule 11.1 (Req)

9075 **external symbol *symbol* defined without a prior declaration**
note If a declaration for an object or function is visible when that object or function is defined, a compiler must verify that the declaration and definition are compatible. A lack of prior declaration prevents such checking.
Supports MISRA C 2012 Rule 8.4 (Req)

9076 **cast from *type* to *type* involves a pointer to an incomplete type other than void**
note Conversions involving pointers to incomplete types cause undefined behavior if converted to or from a floating point type and can cause incorrect alignment if to or from a pointer type. This message is suppressed if the conversion is to void. Also, for purposes of this diagnostic, pointers to void are not treated as pointers to incomplete types.
Supports MISRA C 2012 Rule 11.2 (Req)

9077 **missing unconditional break from final switch case**
note A case at the end of a switch had no unconditional `break`. Some coding guidelines require the use of a `break` for every switch case, including the last one, for maintenance reasons. Note that this message is issued even if the case contains an unconditional `return` statement.
Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C 2012 Rule 16.3 (Req)

9078 **conversion between object pointer type *type* and integer type *type***
note A conversion between a pointer type and an integer/`enum` type was seen. Such conversions can result in undefined behavior if the pointer value cannot be represented in the integer/`enum` type. This diagnostic is not given for null pointer constants.
Supports MISRA C 2012 Rule 11.4 (Adv)

9079 **conversion from pointer to void to pointer to *type***
note Conversion of a pointer to void into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behavior.
Supports MISRA C 2012 Rule 11.5 (Adv)

9080 **integer null pointer constant is not the NULL macro**
note An integer null pointer constant other than the NULL macro was used. Using the NULL macro makes it clear a null pointer constant was intended.
Supports MISRA C 2012 Rule 11.9 (Req)

9081 **too few independent cases for switch**
note A `switch` was seen with fewer than two non-consecutive case labels. A `switch` with fewer than two such cases is redundant and may indicate a programming error.
Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C 2012 Rule 16.6 (Req)

9082 **switch should begin or end with default**
note Placing the `default` label either first or last makes locating it easier.
Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C 2012 Rule 16.5 (Req)

9083 **undefined macro name '*name*' is reserved to the compiler**
note A `#undef` was seen applied to an identifier given by *name* and that identifier is reserved to the compiler by the ISO C/C++ standards.
Supports MISRA C 2004 Rule 20.1 (Req)
Supports MISRA C 2012 Rule 21.1 (Req)

9084 **result of assignment operator used**
note An assignment expression was seen inside a larger expression. The use of assignment operators, simple or compound, in combination with other arithmetic operations can significantly impair the readability of the code.
Supports MISRA C 2012 Rule 13.4 (Adv)
Supports MISRA C++ Rule 6-2-1 (Req)

9085 **statement or comment should appear in default case**
note A `default` label was seen without a comment or statement between it and either the corresponding `break` or, if default is the last case in the switch, the closing `}`. Adding a statement to take action or adding a comment to explain why no action is taken is a form of defensive programming.
Supports MISRA C 2012 Rule 16.1 (Req)
Supports MISRA C 2012 Rule 16.4 (Req)

9087 **cast from pointer to object type (*type*) to pointer to different object type (*type*)**
note A cast was seen between two pointer types that differ with respect to what those types point to. Additionally, the type to which the expression was cast is not a pointer to `char`, whether signed or unsigned. At least one standards organization has cautioned against such a practice.
Supports MISRA C 2004 Rule 11.4 (Adv)
Supports MISRA C 2012 Rule 11.3 (Req)

9088 **named signed single-bit bitfield**
note A named bit-field was declared with a signed data type and only one bit of width. According to the ISO C Standard, a single-bit signed bit-field has one sign bit and no value bits and, consequently does not specify a meaningful value.
Supports MISRA C 2004 Rule 6.5 (Req)
Supports MISRA C 2012 Rule 6.2 (Req)
Supports MISRA C++ Rule 9-6-4 (Req)

9089 **potential side-effect in argument to sizeof**
note An argument to `sizeof` was found to have a potential side-effect. Arguments to `sizeof` are not usually evaluated, unless the argument names a variable length array type. Avoidance of such arguments is advised.
Supports MISRA C 2004 Rule 12.3 (Req)
Supports MISRA C 2012 Rule 13.6 (Mand)

9090 **switch case lacks unconditional break or throw**
note A `switch` case was seen that did not conclude with an unconditional break. Some authors advise against such absences on the grounds they are often errors.
Supports MISRA C 2004 Rule 15.2 (Req)
Supports MISRA C 2012 Rule 16.3 (Req)
Supports MISRA C++ Rule 6-4-5 (Req)

9091 **casting from pointer type *type* to integer type *type***
note A cast of a pointer to an integer type was seen. Since the size of the integer required when a pointer is converted to an integer is implementation defined, some coding guidelines advise against such casts.
Supports MISRA C++ Rule 5-2-9 (Adv)

9093 **the name '*name*' is reserved to the compiler**
note A symbol was delcared with a name reserved to the compiler.
Supports MISRA C 2004 Rule 20.2 (Req)
Supports MISRA C++ Rule 17-0-2 (Req)

9094 **return type of function *symbol* has type alias name difference with previous declaration (*type* vs *type*)**
note This message is similar to 9073 (which deals with parameter types) but applies to return types. In a declaration of a function, the return type specified, while technically identical, uses a different type name than was used for a previous declaration. For example:

```
typedef int INT;
int foo(void);
INT foo(void) {
    ...
}
```

will yield this message.
Supports MISRA C 2004 Rule 8.3 (Req)
Supports MISRA C 2012 Rule 8.3 (Req)
Supports MISRA C++ Rule 3-9-1 (Req)

9095 **symbol *symbol* has same name as previously defined macro**
note A symbol was defined with the same name as a macro that was defined earlier in the same translation unit. For example:

```
#define sum(x, y) ((x)+(y))
int sum = 0;
```

will produce:

```
note 9095: symbol 'sum' has same name as previously defined macro
```

A supplemental message (891) provides the location of the offending macro definition. Note that the message is issued regardless of whether the macro definition is active at the point in which the symbol is declared. For example:

```
#define A
#undef A
int A = 0;
```

will elicit the same complaint for the declaration of `A`.
Supports MISRA C 2012 Rule 5.5 (Req)

9096 **symbol *symbol* has same name as subsequently defined macro**
note This message is similar to `9095` but is issued for symbols defined with the same name as a macro whose definition appears *after* the declaration of the symbol. For example:

```
int A;
#define A 10
```

Unlike message 652, this message will be issued even if the macro is defined outside the scope of the symbol. For example:

```
void foo(int x) { }
#define x 10
```

will not result in a 652 warning since the definition of the `x` macro is outside the scope of the function parameter but `9096` will still be issued.

A supplemental message (891) provides the location of the offending macro definition.
Supports MISRA C 2012 Rule 5.5 (Req)

9097 unparenthesized argument to sizeof operator
note An unparenthesized expression was used as the argument to the `sizeof` operator. While legal, it can result in confusion when used within a larger expression, e.g.:

```
size = sizeof x + y;
```

was this meant to be `sizeof(x) + y` or `sizeof(x + y)`? Using parenthesis can eliminate such questions.
Supports MISRA C 2012 Rule 12.1 (Adv)

9098 pointer argument *integer* (of type *type*) to function *symbol* does not point to a pointer type or an essentially signed, unsigned, boolean, or enum type
note The first or second argument to `memcmp` (or a function with semantics copied from `memcmp`) was not either 1) a pointer to a pointer or 2) a pointer to a MISRA C 2012 essentially signed, unsigned, boolean, or enum type.
Supports MISRA C 2012 AMD1 Rule 21.16 (Req)

9102 possible digraph sequence: '*string*'
note A possible digraph was seen. At least one set of coding guidelines advises against such due to the risk of failure to meet developer expectations.
Supports MISRA C++ Rule 2-5-1 (Adv)

9103 identifier '*name*' with static storage is reused
note An identifier of the given name was seen declared static in one location and not static in another. Some coding guidelines advise against such practice due to the potential for programmer confusion.
Supports MISRA C++ Rule 2-10-5 (Adv)

9104 octal escape sequence used
note Octal escape sequences can be problematic because the inadvertent introduction of a decimal digit (i.e. 8 or 9) ends the octal escape and introduces another character. This diagnostic is not given for `\0`.
Supports MISRA C 2004 Rule 4.1 (Req)
Supports MISRA C 2004 Rule 7.1 (Req)
Supports MISRA C++ Rule 2-13-2 (Req)

9105 unsigned octal and hexadecimal literals require a 'U' suffix
note The inclusion of such a suffix makes clear the value has unsigned type.
Supports MISRA C++ Rule 2-13-3 (Req)

9106 **lower case literal suffix, '*string*'**
note Using upper case literal suffixes removes the potential for ambiguity with respect to literal values.
Supports MISRA C++ Rule 2-13-4 (Req)

9107 **header cannot be included in more than one translation unit because of the definition of symbol *symbol***
note One set of guidelines advises the use of headers in such a way as to avoid the definition of objects or functions that occupy storage.
Supports MISRA C 2004 Rule 8.5 (Req)
Supports MISRA C++ Rule 3-1-1 (Req)

9108 **function *symbol* declared at block scope**
note A function declared at block scope will refer to a member of the enclosing namespace. Additionally, where a declaration statement could either declare a function or an object, the compiler will choose to declare the function. Declaring the function at file scope reduces the likelihood of confusion in both cases.
Supports MISRA C 2004 Rule 8.6 (Req)
Supports MISRA C++ Rule 3-1-2 (Req)

9110 **bit representation of a floating point type used**
note The under lying bit representation of floating point values can differ from compiler to compiler, making reliance upon such representation non-portable.
Supports MISRA C 2004 Rule 12.12 (Req)
Supports MISRA C++ Rule 3-9-3 (Req)

9111 **boolean expression used with non-permitted operator '*string*'**
note The use of expressions of `bool` with certain operators, such as the bitwise operators, is not likely to be either meaningful or intended.
Supports MISRA C++ Rule 4-5-1 (Req)

9112 **plain character expression used with non-permitted operator '*string*'**
note With the exception of the sequence of character values representing 0 thru 9, the exact value of any other particular character is not guaranteed and reliance upon such an order is non-portable.
Supports MISRA C++ Rule 4-5-3 (Req)

9113 **dependence placed on C++ operator precedence**
note The use of parentheses instead of relying upon operator precedence can help make the code easier to understand.
Supports MISRA C++ Rule 5-0-2 (Adv)

9114 **implicit conversion of integer cvalue expression**
note A prominent coding standard has defined the notion of a `cvalue` expression and, to help ensure operations in a given expression are performed within a particular fashion, the guidelines caution against such a value undergoing implicit conversions.
Supports MISRA C++ Rule 5-0-3 (Req)

9115 **implicit conversion from integer to floating point type**
note Such conversions between these two types of values can result in inexact representation.
Supports MISRA C++ Rule 5-0-5 (Req)

9116 **implicit conversion of floating point cvalue expression**
note A prominent coding standard has defined the notion of a `cvalue` expression and, to help ensure operations in a given expression are performed within a particular fashion, the guidelines caution against such a value undergoing implicit conversions.
Supports MISRA C++ Rule 5-0-3 (Req)

9117 **implicit conversion from *underlying-type* to *underlying-type* changes signedness of underlying**
note **type**
Some such conversions can lead to implementation defined behavior. Reliance upon such behavior is, therefore, not portable.
Supports MISRA C++ Rule 5-0-4 (Req)

9118 **implicit conversion from floating point to integer type**
note Such conversions between these two types of values can result in undefined behavior.
Supports MISRA C++ Rule 5-0-5 (Req)

9119 **implicit conversion of integer to smaller type**
note A conversion was performed from an integer to a type that has a smaller MISRA C++ *underlying type*.
Supports MISRA C++ Rule 5-0-6 (Req)

9120 **implicit conversion of floating point to smaller type**
note A conversion was performed from a floating point type to a type that has a smaller MISRA C++ *underlying type*.
Supports MISRA C++ Rule 5-0-6 (Req)

9121 **cast of cvalue expression from integer to floating point type**
note

A cast was used to convert a MISRA C++ `cvalue` expression from an integral to floating point type.
Supports MISRA C++ Rule 5-0-7 (Req)

9122 **cast of cvalue expression from floating point to integer type**
note A cast was used to convert a MISRA C++ `cvalue` expression from a floating point to integral type.
Supports MISRA C++ Rule 5-0-7 (Req)

9123 **cast of integer cvalue expression to larger type**
note A cast was used to convert a MISRA C++ `cvalue` expression of integral type to a type with a larger *underlying type*.
Supports MISRA C++ Rule 5-0-8 (Req)
Supports MISRA C++ Rule 16-2-2 (Req)

9124 **cast of floating point cvalue expression to larger type**
note A cast was used to convert a MISRA C++ `cvalue` expression of floating point type to a type with a larger *underlying type*.
Supports MISRA C++ Rule 5-0-8 (Req)

9125 **cast of integer cvalue expression changes signedness**
note A cast was used to convert a MISRA C++ *cvalue* expression of integral type to an *underlying type* with a different signedness.
Supports MISRA C++ Rule 5-0-9 (Req)

9126 **the result of the *operator* operator applied to an object with an underlying type of *underlying-type* must be cast to *type* in this context**
note The `~` or `<<` operator was applied to an operand with a MISRA C++ *underlying type* of `unsigned char` or `unsigned short` but the result was not cast to the appropriate underlying type.
Supports MISRA C++ Rule 5-0-10 (Req)

9128 **plain character data mixed with non-plain-character data**
note A prominent standard urges, since whether plain `char` is signed or unsigned is implementation defined, the `char` type not be mixed with other types.
Supports MISRA C 2004 Rule 6.1 (Req)
Supports MISRA C 2004 Rule 6.2 (Req)
Supports MISRA C++ Rule 5-0-11 (Req)

9130 **bitwise operator '*operator*' applied to signed underlying type**
note The specified bitwise operator was applied to an operand with a signed MISRA C++ *underlying type*.
Supports MISRA C++ Rule 5-0-21 (Req)

9131 ***left/right* side of logical operator '*operator*' is not a postfix expression**
note Using only postfix-expressions with logical operators helps to improve readability of the code. Note this message is not given if the expression consists of either a sequence of only logical `&&` or a sequence of only logical `||`.
Supports MISRA C++ Rule 5-2-1 (Req)

9132 **array type passed to function expecting a pointer**
note Array-to-pointer decay results in a loss of array bound information. A function depending upon an array to have a certain length, if that array decays to a pointer, can result in out-of-bounds operations, depending upon whether or not the bound of the original array matches with expectations.
Supports MISRA C++ Rule 5-2-12 (Req)

9133 **boolean expression required for operator '*string*'**
note The use of non-bool operands with `!`, `&&`, or `||` is unlikely to be meaningful or intended. A more likely scenario is the programmer meant to use such an operand with one of the bitwise operators.
Supports MISRA C++ Rule 5-3-1 (Req)

9134 **unary minus applied to operand with unsigned underlying type**
note A unary minus was applied to an expression with a signed MISRA C++ *underlying type*.

Supports MISRA C++ Rule 5-3-2 (Req)

9135 **unary operator & overloaded**
note The unary operator `&` was overloaded.
Supports MISRA C++ Rule 5-3-3 (Req)

9136 **the shift value is at least the precision of the MISRA C++ underlying type of the left hand side**
note The value specified for the right hand side of a shift operator was out of bounds for the MISRA C++ *underlying type* on the left hand side of the operator.
Supports MISRA C++ Rule 5-8-1 (Req)

9137 **testing floating point values for equality**
note A floating point value was tested, directly or indirectly, for (in)equality with another value.
Supports MISRA C++ Rule 6-2-2 (Req)

9138 **null statement not on line by itself**
note A null statement was encountered that, before preprocessing, did not appear on a line by itself. Comments following the null statement are allowed as long as there is whitespace separating the null statement from the comment.
Supports MISRA C 2004 Rule 14.3 (Req)
Supports MISRA C++ Rule 6-2-3 (Req)

9139 **case label follows default in switch statement**
note A `case` label was encountered following the `default` label of a `switch` statement.
Supports MISRA C 2004 Rule 15.3 (Req)
Supports MISRA C++ Rule 6-4-6 (Req)

9141 **global declaration of symbol *symbol***
note The specified symbol was declared in the global namespace.
Supports MISRA C++ Rule 7-3-1 (Req)

9142 **function main declared outside the global namespace**
note A function with the name '`main`' was declared that was not the global main function.
Supports MISRA C++ Rule 7-3-2 (Req)

9144 **using directive used: '*string*'**
note A using directive was encountered.
Supports MISRA C++ Rule 7-3-4 (Req)

9145 **using *declaration/directive* in header '*file*'**
note A using directive or using declaration was encountered in a header file. This message is not issued for using declarations in class or function scope.
Supports MISRA C++ Rule 7-3-6 (Req)

9146 **multiple declarators in a declaration**
note A declaration was encountered that contains multiple declarators. For example:

```
int i, j;
```

will elicit this message.
Supports MISRA C++ Rule 8-0-1 (Req)

9147 **implicit function-to-pointer decay**
note The unadorned name of a function was encountered that was not part of a function call.
Supports MISRA C 2004 Rule 16.9 (Req)
Supports MISRA C++ Rule 8-4-4 (Req)

9148 **'=' should initialize either all enum members or only the first for enumerator *symbol***
note Unintentional duplication of enumerator values can occur when an enumeration consists of members with explicit and implicit values.
Supports MISRA C 2004 Rule 9.3 (Req)
Supports MISRA C++ Rule 8-5-3 (Req)

9149 **bit field must be explicitly signed integer, unsigned integer, or bool**
note When using 'int' or 'wchar_t' as the bit-field type, it is implementation defined whether or not the type used is a signed type. Explicitly specifying 'signed' or 'unsigned' makes it clear what type will be used as the underlying type.
Supports MISRA C 2012 Rule 6.1 (Req)
Supports MISRA C++ Rule 9-6-2 (Req)
Supports MISRA C++ Rule 9-6-3 (Req)

9150 **non-private data member *symbol* within a non-POD structure**
note A member of a non-POD structure was declared public or protected.
Supports MISRA C++ Rule 11-0-1 (Req)

9151 **abstract class *symbol* declares public copy assignment operator *symbol***
note A public copy assignment operator was declared in an abstract class.
Supports MISRA C++ Rule 12-8-2 (Req)

9153 **viable set contains both function *symbol* and template *symbol***
note In a context where a name resolves either to a non-template function or to a specialization of a function template (typically a call), the set of viable candidates included both.
Supports MISRA C++ Rule 14-8-2 (Adv)

9154 **throwing a pointer**
note A pointer type was passed to a `throw` expression. It may not be clear who is responsible for cleaning up the pointed to object.
Supports MISRA C++ Rule 15-0-2 (Adv)

9156 **rethrow outside of catch block will call std::terminate if no exception is being handled**
note An empty throw expression was encountered outside of a try-catch block. An empty throw re-throws the

currently handled exception. If there is no such exception `std::terminate()` will be called. This is likely to be unintended.
Supports MISRA C++ Rule 15-1-3 (Req)

9158 #define used within *string symbol* for macro '*name*'
note A macro was defined inside the braced region of the entity described by *string* (such as `function` or `class`). Such usage could imply the belief that the scope of the macro definition is limited to the braced region, which is not the case.
Supports MISRA C 2004 Rule 19.5 (Req)
Supports MISRA C++ Rule 16-0-2 (Req)

9159 #undef used within *string symbol* for macro '*name*'
note A macro was undefined inside the braced region of the entity described by *string* (such as `function` or `class`). Such usage could imply the belief that the scope of the directive is limited to the braced region, which is not the case.
Supports MISRA C 2004 Rule 19.5 (Req)
Supports MISRA C++ Rule 16-0-2 (Req)

9160 unknown preprocessor directive '*string*' in conditionally excluded region
note Within a conditionally excluded region, a line that started with a `#` was seen but was not part of a valid preprocessing directive. Error 16 is produced if an unknown preprocessor directive appears in a non-excluded region.
Supports MISRA C 2004 Rule 19.16 (Req)
Supports MISRA C 2012 Rule 20.13 (Req)
Supports MISRA C++ Rule 16-0-8 (Req)

9162 use of '*string*' at global scope
note Either a `static_assert()` or a using-declaration was seen at global scope, as indicated by the *string*.
Supports MISRA C++ Rule 7-3-1 (Req)

9165 function *symbol* defined with a variable number of arguments
note The named function is defined to take a variable number of arguments. At least one author advises against such a practice because doing so avoids the type checking provided by the compiler.
Supports MISRA C 2004 Rule 16.1 (Req)
Supports MISRA C++ Rule 8-4-1 (Req)

9167 macro '*name*' defined in *string symbol* not undefined in same *string*
note A macro was defined inside of a declaration of a `function`, `class/struct/union`, `namespace`, or `enumeration` and was not undefined within the braced region of that declaration. The macro will persist beyond the end of the declaration, which may not be intended. For example:

```
void foo() {
#define A ...

}
```

will result in the message:

```
macro 'A' defined in function 'foo' not undefined in same function
```

9168 **variable *symbol* has type alias name difference with previous declaration (*type* vs *type*)**
note A variable is declared in two places with types that, while technically identical, have different alias names. For example:

```
typedef int INT;
extern int var;
INT var;            // note 9168
```

Supports MISRA C++ Rule 3-9-1 (Req)

9169 **constructor *symbol* can be used for implicit conversions from fundamental type *type***
note A constructor was found that could be used for implicit conversions from a fundamental type. This message is similiar to `1931` but only reports instances where the implicit conversion is from a fundamental type (e.g. integer and floating point types but not pointers, references, arrays, classes, etc.). Like message `1931`, if the constructor is declared with the keyword `explicit`, this message will not be emitted. This message is also not be emitted for variadic constructors.
Supports MISRA C++ Rule 12-1-3 (Req)

9170 **pure function *symbol* overrides non-pure function *symbol***
note The specified function is declared as pure but overrides a non-pure function in a base class. Was this a mistake?
Supports MISRA C++ Rule 10-3-3 (Req)

9171 **downcast of polymorphic type *type* to type *type***
note A cast was used to convert a pointer to a polymorphic type (a class that contains or inherits one or more virtual functions) to a pointer to a derived class.
Supports MISRA C++ Rule 5-2-3 (Adv)

9172 **bitwise operator '*operator*' used with non-constant operands of differing underlying types**
note A bitwise operator was used whose operands did not have the same MISRA C++ *underlying type*. This message is not produced if either operand is an integer constant expression.
Supports MISRA C++ Rule 5-0-20 (Req)

9173 **use of non-placement allocation function *symbol***
note The use of `new` or `delete` was encountered that will allocate or deallocate dynamic memory. Placement `new` is not reported as it does not allocate memory.
Supports MISRA C++ Rule 18-4-1 (Req)

9174 ***type* is a virtual base class of *symbol***
note A class derivation was marked as virtual; some coding standards prohibit virtual inheritance due to the potential complexities involved.
Supports MISRA C++ Rule 10-1-1 (Adv)

9175 **function *symbol* has void return type and no external side-effects**
note The specified function does not appear to have any external side-effects and does not return any information so what is the purpose of calling the function?
Supports MISRA C++ Rule 0-1-8 (Req)

9176 **pointer type *type* converted to unrelated pointer type *type***
note A pointer was converted (implicitly or explicitly) to a different pointer type and the source pointee type was not a class or structure derived from the destination pointee type.
Supports MISRA C++ Rule 5-2-7 (Req)

9204 **hexadecimal escape sequence used**
note A hexadecimal escape sequence (`\x`) was used inside a character or string literal.
Supports MISRA C 2004 Rule 4.1 (Req)

9209 **plain character data used with prohibited operator *string***
note The plain `char` type is defined by the implementation to have the same size and range as either `signed char` or `unsigned char` but is a separate and distinct type. For this reason, it is often recommended that `char` be used for character data and `signed char` and `unsigned char` be used for numeric data. This message reports when an object of plain `char` type is used as an operand to a unary operator or a binary operator other than `=`, `==`, and `!=`.
Supports MISRA C 2004 Rule 6.1 (Req)

9212 **bit field type *type* is not explicitly signed int or unsigned int**
note A bit field was defined with a type other than `signed int` or `unsigned int` or with a `typedef` that is defined using one of these two explicit types. When using plain `int` as a bit-field type, the signedness of the type used is implementation defined. Only `int` (`signed` and `unsigned`) and `_Bool` (in C99) are sanctioned for use in bit-fields, use of any other type results in implementation-defined behavior. `-etype(9212, _Bool)` can be used to suppress this message for the C99 `_Bool` type. See also message 9149, which is similar but more lenient.
Supports MISRA C 2004 Rule 6.4 (Req)

9224 **expression is not effectively boolean and must be explicitly tested for zero**
note An expression that is not "effectively boolean" is being implicitly tested for zero in the controlling expression of an `if` statement, an iteration statement, or the first operand of a conditional operator. For example, given that `x` is an integer:

```
if (x)
```

will elicit this message while:

```
if (x != 0)
```

will not. "Effectively boolean" value are produced by the operators `==`, `!=`, `<=`, `>=`, `<`, `>`, `!`, `||`, and `&&`.
Supports MISRA C 2004 Rule 13.2 (Adv)

9225 **integral expression of underlying type *underlying-type* cannot be implicitly converted to type**
note ***type* because it is not a wider integer type of the same signedness**
An integral expression was implicitly converted from to a *type* that was not a wider *type* of the same signedness.
Supports MISRA C 2004 Rule 10.1 (Req)

9226 **integral expression of underlying type *underlying-type* cannot be implicitly converted to type**
note ***type* because it is *string***
A complex integral expression was implicitly converted to a different *type* or a non-constant integral expression was implicitly converted to a different *type* while being passed to or returned from a function. "Complex"

here means an expression that is not an `lvalue` and is not a function return value.
Supports MISRA C 2004 Rule 10.1 (Req)

9227 note **floating expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is not a wider floating type**
A floating point expression was implicitly converted to a *type* that is not a wider *type*.
Supports MISRA C 2004 Rule 10.2 (Req)

9228 note **floating expression of underlying type *underlying-type* cannot be implicitly converted to type *type* because it is *string***
A floating point expression was implicitly converted to a different type in a context in which a cast should be used to be compliant with MISRA C 2004. The context is provided in *string*, which is one of a `complex expression`, a `function argument`, or a `return value`.
Supports MISRA C 2004 Rule 10.2 (Req)

9229 note **complex integral expression may only be cast to another integral type of the same signedness no wider than the original type**
A complex expression with integral type was cast to a type with different signedness or whose *underlying type* is wider than the *underlying type* of the expression.
Supports MISRA C 2004 Rule 10.3 (Req)

9230 note **complex floating expression may only be cast to another floating type no wider than the original type**
A complex expression with floating point type was cast to a type with whose *underlying type* is wider than the *underlying typ*e of the expression.
Supports MISRA C 2004 Rule 10.4 (Req)

9231 note **result of *operator* operator applied to operand of type *type* must be immediately cast to *type***
The `~` or `<<` operator was applied to an operand with a MISRA C *underlying type* of `unsigned char` or `unsigned short` but the result was not cast to the appropriate underlying type.
Supports MISRA C 2004 Rule 10.5 (Req)

9232 note **expected/did not expect an effectively boolean argument for operator *operator***
A MISRA C *effectively boolean* expression was used as an operand to an operator that should not operate on such an expression or an operator for which an *effectively boolean* expression was expected was not provided one. Specifically, the operators `&&`, `||`, and `!` should contain only `effectively boolean` operands and `effectively boolean` operands should not be used with operators other than `&&`, `||`, `!`, `=`, `==`, `!=`, and `?:`.
Supports MISRA C 2004 Rule 12.6 (Adv)

9233 note **bitwise operator *operator* may not be applied to operand with signed underlying type**
An expression with a MISRA C signed *underlying type* was provided as an operand to a bitwise operator.
Supports MISRA C 2004 Rule 12.7 (Req)

9234 note **shift amount exceeds size of operand's underlying type**
An expression was shifted by a negative amount or an amount greater than the bit width of the expression's MISRA C *underlying type*.

Supports MISRA C 2004 Rule 12.8 (Req)

9235 unary minus applied to operand with unsigned underlying type
note The unary minus operator was applied to an expression with an unsigned MISRA C *underlying type*.
Supports MISRA C 2004 Rule 12.9 (Req)

9236 assignment operator may not be used within a boolean-valued expression
note An assignment operator was used within a boolean context, such as comparing the result of assignment to a specific value.
Supports MISRA C 2004 Rule 13.1 (Req)

9237 conversion between pointer to function type *type* and differing non-integral type *type*
note A conversion was performed between a pointer to function and a pointer to a different type that was not a pointer to an integral type.
Supports MISRA C 2004 Rule 11.1 (Req)

9238 switch condition may not be boolean
note The conditional expression of a `switch` statement has a MISRA C *effectively boolean* type.
Supports MISRA C 2004 Rule 15.4 (Req)

9240 *left/right* side of logical operator '*operator*' is not a primary expression
note This message is issued when the operands of the `||` and `&&` operators are not *primary-expressions*.
Supports MISRA C 2004 Rule 12.5 (Req)

9252 testing floating point for equality using exact value
note Message 777 is issued when an object with a floating point type is tested for equality using either `==` or `!=`. Message 777 is not issued when one of the operands is a value that can be represented exactly in the corresponding floating point representation, such as 0 or 13.5. In such cases, this message is issued instead.
Supports MISRA C 2004 Rule 13.3 (Req)

9254 continue statement encountered
note A `continue` statement was seen. Some coding guidelines forbid the use of `continue` statements.
Supports MISRA C 2004 Rule 14.5 (Req)
Supports MISRA C++ Rule 6-6-3 (Req)

9259 C comment contains '://' sequence
note Message 9059 reports on cases where a C comment contains what may be a C++ comment, e.g. the sequence '//'. Because including URLs inside of comments is a common practice, message 9059 is not issued when the '//' sequence is immediately preceded by a ':' to prevent the message from being issued in cases such as:

```
/* See http://www.gimpel.com for details */
```

This message fills the gap by reporting on the instances not reported by 9059.
Supports MISRA C 2012 Rule 3.1 (Req)

9260 C++ style comment used
note A C++-style comment (//) was encountered. Such comments were not part of C until C99 and may not be

consistently supported by older compilers.
Supports MISRA C 2004 Rule 2.2 (Req)

9264 note **array subscript applied to variable *symbol* declared with non-array type *type***
The base of an array subscript operation was not declared as an array (i.e., it was declared as a pointer). Some coding guidelines suggest that array subscript operations should only be applied to array types.
Supports MISRA C 2004 Rule 17.4 (Req)

9272 note **parameter *integer* of function *symbol* has different name than overridden function *symbol* (*symbol* vs *symbol*)**
This message is similar to 9072 but applied to differences between overridden functions. In the declaration of a function, the *name* given to the specified parameter is different from the *name* given for the same parameter in the declaration of one of the functions being overridden. For example:

```
struct A {
    virtual void foo(int width);
};
struct B : A {
    void foo(int depth);
};
```

will yield this message because `A::foo` uses width as the name of the first parameter while the overridden function `B::foo` uses the name depth.
Supports MISRA C++ Rule 8-4-2 (Req)

9273 note **parameter *integer* of function *symbol* has type alias name difference with overridden function *symbol* (*type* vs *type*)**
This message is similar to 9073 but applies to differences between overridden functions. In the declaration of a function, the specified parameter is declared with a *type* that, while technically identical, uses a different *name* for the *type* than was used for the parameter in the declaration of one of the functions that this function overrides. For example:

```
typedef int INT;

struct A {
    virtual void foo(int);
};

struct B : A {
    void foo(INT);
};
```

will yield this message because `A::foo` is declared with a first parameter of type `int` while the overridden function `B::foo` declared the parameter with type `INT`, a type alias name difference.

9287 note **cast from pointer to object type (*type*) to pointer to char type (*type*)**
A cast was performed between a pointer to an object type and a pointer to a character type. The actual pointer types are provided in the message.
Supports MISRA C 2004 Rule 11.4 (Adv)

9288 **unnamed signed single-bit bitfield**
note An unnamed signed bit-field was declared with a single bit.
Supports MISRA C 2004 Rule 6.5 (Req)

9294 **return type of function *symbol* has type alias name difference with overridden function *symbol* (*type* vs *type*)**
note This message is similar to 9094 but applies to differences between overridden functions. In a declaration of a function, the return type specified, while technically identical, uses a different type name than was used for the declaration of one of the functions that this function overrides. For example:

```
typedef int INT;

struct A {
    virtual int foo();
};

struct B : A {
    INT foo();
};
```

will yield this message.

9295 **conversion between object pointer type *type* and non-integer arithmetic essential type '*essential-type*'**
note This message is issued when a cast is used to convert between a pointer to object type and a non-integer arithmetic MISRA C 2012 essential type. For example:

```
enum color { RED, GREEN, BLUE };
void foo(int *pi) {
    enum color c = (enum color)pi;  // Note 9295
}
```

Supports MISRA C 2012 Rule 11.7 (Req)

9401 **function *symbol* returns pointer to void**
note The specified function returns a pointer to void, which some consider to be unsafe because it can compromise type safety.

9402 **function *symbol* parameter *integer* is void pointer**
note The specified function accepts a void pointer as an argument, which some consider to be unsafe because such pointers can compromise type safety.

9403 **function *symbol* parameter *integer* has same unqualified type (*type*) as previous parameter**
note A function has two consecutive parameters of the same (unqualified) type. Functions that accept many arguments can be difficult to use correctly as the chances of misordered arguments increases as the number of parameters increase. When arguments are of different types, misordered arguments are more likely to be caused by the compiler. When consecutive parameters are of the same type, calls to the function that accidentally transpose the arguments are less likely to be noticed.

9404 **destructor for class *symbol* should be declared 'noexcept'**
note Given that destructors should never `throw`, declaring them as '`noexcept`' is wise as it allows the compiler to ensure this is the case.

9405 **move constructor for class *symbol* should be declared 'noexcept'**
note Move constructors should not `throw`; declaring them as '`noexcept`' allows the compiler to ensure this is the case.

9406 **move assignment operator *symbol* should be declared 'noexcept'**
note Move assignment functions should not `throw`; declaring them as '`noexcept`' allows the compiler to ensure this is the case.

9407 **copy assignment operator *symbol* should not be virtual**
note A copy assignment operator was declared as virtual; this is rarely the right thing to do.

9408 **copy assignment operator *symbol* should take a const reference type**
note A copy assignment operator should take a `const` reference argument.

9409 **copy assignment operator *symbol* should return a non-const lvalue-reference type**
note A copy assignment operator should return a non-`const` `lvalue`-reference type.

9410 **move assignment operator *symbol* should not be virtual**
note A move assignment operator was declared as `virtual`, this is rarely the right thing to do.

9411 **move assignment operator *symbol* should take a non-const reference type**
note A move assignment operator was declared whose argument is not a non-`const` reference.

9412 **move assignment operator *symbol* should return a non-const lvalue-reference type**
note A move assignment operator was declared that doesn't return a non-`const` `lvalue`-reference type.

9413 **class *symbol* contains data members of differing access levels**
note A class contains data members declared with different access levels.

9416 **typedef used to define name *symbol***
note A `typedef` was used to define a type alias instead of a using alias.

9417 **data member *symbol* has protected access level**
note The specified *data* member of a class has an access level of `protected`. Some authors suggest against using `protected` data members.

9901 **return value '*string*' for call to function *symbol* updated to '*string*' via return semantic '*string*'**
note This message is emitted when a return semantic adds or updates value tracking information for a return

value of a function call either because the semantic contains more specific information than was gleaned from walking the body of the called function or because the fso flag was set.

9902 **return value '*string*' for call to function *symbol* not updated by return semantic '*string*' which adds no new information**
note
This message is emitted when a return semantic is not applied to a function call because the semantic does not provide any information that was not already known from walking the call.

9903 ***essential-type-preview***
note
This message shows the step-by-step evaluation of how an expression's MISRA C 2012 *essential type* is calculated. See the f12 flag for details.

9904 **hook event: '*string*'**
note
This message is emitted every time a hookable event is reached in the AST walking phase.

9905 **value tracking debug assertion not known to be unequivocally true**
note

16 Differences from PC-lint 9.0

Note: This section describes differences between PC-lint 9.0 and the initial release of PC-lint Plus 1.0.

- PC-lint Plus contains many more diagnostics than PC-lint. As a result, the range of message numbers has increased. In particular, C language diagnostics extend into the `2000-2999` range, C++ diagnostics extend into the `3000-3999` range, the range `4000-5999` is used for new general error messages, and the ranage 8000-8999 are reserved for user-defined messages.. See 15 Messages for complete details.

  Only message suppression options that appear before the location specified in the message are considered, even if the actual message is not issued until later (such as at global wrapup).

- K&R (traditional, pre-ANSI) C is no longer supported. In particular, options and flags which were specific to K&R C will not be supported. We will continue to support C89/C90, C99, and C11.

- The default diagnostic format has changed. By default, we now display the message first, followed by the corresponding source line. We also use a caret (`^`) to indicate source positions by default instead of an underscore. Finally, tildes (`~`) are used to underscore relevant parts of source code. The default PC-lint 9.0 format can be obtained using the option `-ha_3`.

- Macro display has changed. When a diagnostic is issued from a location which is the result of a macro expansion, a separate `893` note message will be attached to the original message for each expansion associated with the message.

- The message category emitted with each message (error, warning, info, note and supplemental) is now displayed in all lowercase letters by default which is a departure from PC-lint which capitalized the first letter. The new `+fcc` option can be used to revert to the previous behavior.

- The `-c` option is no longer the preferred way to configure PC-lint Plus for a particular compiler. The problem is that it is not always clear exactly what the `-c` option does for a particular compiler and compiler versions cannot be specified with the `-c` option. Compiler-specific configuration is supported through the use of compiler `.lnt` files (which has been the case for most compilers for a while anyway).

- The non-standard use of `sizeof` within a preprocessor statement is no longer supported. This has never been allowed by ANSI/ISO C or C++ although it is an extension in some older compilers. A work-around is provided with the `-pp_sizeof` option.

- Constant variables and enumeration constants are not supported inside of user-defined function semantics. Macros are still expanded and environment variables can now be used in function semantic specifications as well.

- The suppression context of the location cited in a message is now taken into consideration when determining whether to suppress the message, despite when the message is issued. For example, given:

  ```
  int x;
  //lint -e714
  int y;
  ```

  PC-lint would not issue message `714` for either `x` or `y` because the message is not issued until wrap up time at which point message `714` is suppressed. In PC-lint Plus, when we consider issuing the message for `x`, we will remember the message suppression state at the point in which `x` was declared (the location provided in the message) and give the message since at that location the message is not suppressed. The message will not be issued for `y` because by this point the context includes the suppression of message `714`. In most cases this results in considerably more intuitive behavior. For example, it is now possible to suppress wrap-up messages using a single-line suppression at the location in which the message is given which was not possible before:

  ```
  int x; //lint !e714
  ```

- Options appearing within source code will no longer have any effect outside of the containing translation unit. In other words, at the end of each module, the option state reverts back to what it was before the module was processed. This is a change from PC-lint where the effect of an option provided in one source file would "leak" into following source files. This was rarely the intended behavior and resulted in situations where analysis would be dependent upon the order in which modules were processed. Note in particular that global options (such as `-u` [unit checkout]) no longer have an effect when appearing inside of source modules.

- The behavior of the precompiled header feature has changed. In particular, multiple PCH files are supported in one project (but only one per module) and the way that PCH files are processed diverges from PC-lint 9. See 6.1 Precompiled Headers for more information.

- PC-lint Plus returns zero in the absence of fatal or internal errors, i.e. the default return value no longer reflects the number of messages emitted. The `-frz` option will restore the previous behavior. See 3.2 Exit Code for details.

16.1 Major New Features

- Full support for recent versions of C and C++ including C99, C11, C++11 and C++14.

- We now support Visual Studio 2013, Visual Studio 2015 and Visual Studio 2017.

- Improved support for gcc compiler extensions such as case ranges, locally declared labels, and labels as values.

- Value Tracking contains a number of improvements including structure member and pointer tracking, new and improved diagnostics, and a new architecture that allows for deeper analysis.

- Significantly improved location information provided with diagnostics.

- Improvements to the Semantics feature including user-defined semantics that can be applied to individual function overloads, identification of invalid semantics, and user-defined return semantic validation.

- Strong Type checking and Dimensional analysis provide more detailed information and can suggest corrections.

- Format string checking supports positional arguments and implements several new checks.

- A number of new diagnostics have been added while outdated diagnostics have been removed.

16.2 General Diagnostic Changes

- PC-lint 9.00k introduced the 9xxx message range for additional Elective Notes, currently used mainly for MISRA C 2012 and (since 9.00L) MISRA C++ messages. PC-lint Plus adds the new message ranges 2000-2999 for C diagnostics, 3000-3999 for C++ diagnostics. The complete set of ranges is provided in the below table.

Range	Description	Warning Level
1-199	C Syntax Errors	1
200-299	Internal Errors	1
300-399	Fatal Errors	1
400-699	C Warnings	2
700-899	C Informational	3
900-999	C Elective Notes	4
1000-1199	C++ Syntax Errors	1
1200-1299	Internal Errors	1
1300-1399	C++ Fatal Errors	1
1400-1699	C++ Warnings	2
1700-1899	C++ Informational	3
1900-1999	C++ Elective Notes	4
2000-2199	C Syntax Errors	1
2400-2699	C Warnings	2
2700-2899	C Informational	3
2900-2999	C Elective Notes	4
3000-3199	C++ Syntax Errors	1
3400-3699	C++ Warnings	2
3700-3899	C++ Informational	3
3900-3999	C++ Elective Notes	4
4000-5999	C and C++ Errors	1
8000-8999	User Defined	3
9000-9999	Misc Elective Notes	4

Note: Messages related to C may also appear while processing C++ source but C++ messages should not appear while processing C source code.

- The precision of the location (line and column) associated with most messages has been significantly improved. For example, given the C source:

```
void f() {
    int i;
}
```

PC-lint 9.00L generates the diagnostic:

```
_
}
Warning 529: Symbol 'i' (line 2) not subsequently referenced
```

while PC-lint Plus generates:

```
warning 529: local variable 'i' declared in 'f' not subsequently referenced
int i;
    ^
```

- The verbiage of many existing messages has been clarified and/or elaborated so as to more quickly understand the point of the diagnostic.
- All diagnostics now start with a lowercase letter. Previously, most messages began with an uppercase letter but this was not consistent.
- The message category (error, warning, info, note) is now emitted in all lowercase letters by default. Previously, the first letter was capitalized in messages (e.g. Error vs error). The previous behavior can be obtained using the `+fcc` flag.

- When the location of a message is the result of a macro expansion, this fact is relayed with the new message 893 ("expanded from macro '*string*'"). In PC-lint, this information was relayed using a macro display line that was emitted above the source context line.

16.3 Value Tracking

Major improvements to value tracking include:

- A new value tracking model, which keeps track of more value information and without utilizing multiple passes. The result is often a deeper and more accurate analysis with diagnostics presented in a more lucid fashion.

- Tracking of structure members.

  In the following example, PC-lint previously did not detect a division by 0 due to the general absence of structure member tracking (outside of the `this` object for member functions):

```
struct S { int x; };
int f(int i) {
    S s;
    s.x = 0;
    return i / s.x;
}
```

  While PC-lint Plus issues the following:

```
test.cpp 6 warning 414: division by zero
    return i / s.x;
             ^ ~~~
test.cpp 6 supplemental 831: expression evaluates to 0
    return i / s.x;
               ^~~
```

- Tracking of pointers.

  PC-lint now supports the tracking of pointer values where feasible. In the following example, PC-lint previously could not report a division by `0` message as the indirect modification through `pi` was not recognized as modifying `i`:

```
int f(int x) {
    int i = 1;
    int *pi = &i;
    *pi = 0;
    return x / i;
}
```

  PC-lint Plus reports the issue:

```
warning 414: division by zero
    return x / i;
             ^~
supplemental 831: expression evaluates to 0
return x / i;
           ^
```

  As another example, PC-lint Plus now recognizes the division by `0` in the following example by realizing the call to `x` is really a call to `f` that sets the static variable `i` to `0` by tracking the values of function pointers:

```
static int i = 1;
void f() { i = 0; }

int g() {
    auto x = f;
    x();
    return 1 / i;
}
```

- As the new value tracking system can perform deeper intramodule analysis of specific calls in a single pass, some use cases for the `-passes` option have been supplanted by the new `-vt_depth` option, which is used to specify the maximum call stack depth PC-lint Plus should recurse during specific walks within a module. The default value is 2. A higher value will result in deeper walking during specific walks without resulting in excess memory usage or additional time spent when there are no functions at a higher level (unlike the `-passes` option in previous versions, which always results in memory and CPU increase). The `-passes` option (an alias of the new name `-vt_passes` for consistency) still controls the number of passes in which all modules are re-parsed and intermodule calls are performed.

- Tracking of floating point values.

  PC-lint Plus now tracks floating point values, which extends the previous behavior that tracked only integral values.

16.4 Semantics

The following enhancements have been added to the Semantics feature:

- User-defined return semantics are now validated for specific calls when the body of the function is available. Violations of the return semantic are reported via the new 2426 message. In the following example, the semantic `@p > 0` specifies that the pointer return value is never null. The implementation of this function contains a path that violates this semantic. PC-lint Plus will now report when such a path is taken causing the return value semantic to be violated:

  ```
  //lint -sem(f, @p > 0) return value should never be null
  void *f(int a, void *p) {
      if (a < 0)
          return 0;
      return p;
  }

  void g(void *p) {
      void *ptr = f(-1, p);
  }
  ```

  PC-lint Plus produces:

  ```
  warning 2426: return value (NULL) of call to function 'f(int, void *)'
      conflicts with return semantic '(@p>0)'
      void *ptr = f(-1, p);
                  ^
  ```

  This checking extends user-defined semantics from a set of assumptions about a function where no body exists to a contract validation mechanism when the body is available.

- Separate user-defined semantics are now supported for overloaded functions.

  A user-defined semantic can be applied to a specific function overload by including the function's argument list in the semantic specification. A user-defined semantic that does not include an argument list will apply to all overloads that do not have a semantic associated with the specific overload.

- Certain GCC-style attributes are now (by default) automatically translated to the corresponding function semantics. See the description of the new `fca` flag option for details.

- PC-lint Plus now warns when an invalid user-defined semantic is being rejected for a specific call via the new 2425 message. An explanation of why the semantic is being discarded is provided in the message. For example, the semantic below specifies that the first numeric argument to function `f` should be larger than the second argument. In the actual call, there is no second argument so the semantic doesn't apply:

```
//lint -sem(f, 1n > 2n)
int f(int i);
void g() {
    f(1);
}
```

  PC-lint Plus will now emit the diagnostic:

```
warning 2425: user-defined function semantic '(1n>2n)' was
    rejected during call to function 'f(int)' because '2n'
    references non-existant argument in call
    f(1);
    ^
```

  The warning highlights a likely mistake in the specification of the semantic. If there are multiple overloads and the semantic should only apply to certain overloads, the overload-specific semantic specification described above can be employed.

- PC-lint Plus favors the information in a user-defined return semantic even when more precise information was known about this value. The default behavior is to retain the more specific information. The old behavior can be reinstated using the new `+fso` flag option. This only applies to functions with an implementation visible to PC-lint.

- New Elective Notes 9901 and 9902 specify debugging information related to the information inferred from a return semantic for a specific call.

16.5 Strong Types and Dimensional Analysis

The messages related to strong type violations are more descriptive and can provide suggestions for certain erroneous constructs. In the following example:

```
//lint -strong(JcAc, Mi, Km, MiPerKm = Mi / Km)
typedef double Mi, Km, MiPerKm;
MiPerKm mi_per_km = 0.62137;

Mi earth_radius = 7918;
void f() {
    Km earth_circumference = 2 * 3.14159 * earth_radius;
}
```

PC-lint would previously generate the following diagnostic:

```
-
    Km earth_circumference = 2 * 3.14159 * earth_radius;
Warning 632: Assignment to strong type 'Km' in context: initialization
```

PC-lint Plus now generates the following:

```
warning 632: strong type mismatch: assigning 'Mi' to 'Km' in
    context 'initialization'
```

```
    Km earth_circumference = 2 * 3.14159 * earth_radius;
                             ^~~~~~~~~~~~~~~~~~~~~~~~~~
supplemental 892: did you mean to divide by a factor of type 'MiPerKm'?
```

16.6 Improved Format String Checking

The checking of `printf` and `scanf` format strings has been significantly improved. In particular, PC-lint Plus now recognizes POSIX positional arguments and diagnoses issues related to positional arguments, existing messages have been re-organized and clarified, and new diagnostics have been added.

16.7 Miscellaneous enhancements and Quiet Changes

- The new `-std` option (e.g. `-std=c++14`) is now the preferred way of specifying a standard language version. The supported values for this option are: `c89`/`c90`, `c99`, `c11`, `C++03`, `C++11`/`C++0x` , `C++14`/`C++1y`, and `C++17`/`1z`. Unlike the `-A` option, this option requires one of the above values, e.g. neither `C++2011` nor `13` is a valid way to specify `C++11` support. The `-A` option is still supported but deprecated, users are encouraged to switch to the new `-std` option.
- The default C language version is C11 and the default C++ language version is C++14. In previous versions the defaults were C99 and C++03 respectively. To restore the previous behavior, the options `-std=c99` and `-std=C++03` may be used.
- The `--e{` option now suppresses messages for the entire enclosing braced region, as the manual has always indicated. Previously, it did not apply to messages issued earlier in the braced region.
- A space-separated list of messages may be specified in parenthesized suppression options, e.g. `-esym(123 456 117*, A, B, C)` will suppress messages `123`, `456`, and `1170-1179` when parameterized by the symbol `A`, `B`, or `C`. This was a long-standing undocumented feature until 9.00L when it was inadvertently removed. The feature has been re-instated with official status.
- Environment variables surrounded by `%` are now supported in all options and in `.lnt` files. Previously environment variables were only expanded in a few places.

16.8 Error Inhibition

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

`+group(`*name*` [,`*pattern*`]...)` adds messages to a group
`-group(`*name*` [,`*pattern*`]...)` remove messages from a group or delete a grouping

The `-efile`/`+efile` options now suppress messages *within* the specified files instead of messages *about* the specified files.

16.9 Verbosity

`-verbosify(`*string*`)` print *string* as a verbosity message

16.10 Message Presentation

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

16.11 Miscellaneous Options

16.11.1 Global

`-cond(conditional-expr, true-options [,false-options])` conditionally execute options
`-dump_message_list=filename` dumps PC-lint Plus message list to the provided file
`-help=option display detailed help about a specific option`
`-write_file(string, filename [,append=true|false] [,binary=true|false])` write a *string* to a *filename*
`+zero_err(# [#]...)` specify message numbers that shouldn't increase exit code
`-zero_err(# [#]...)` specify message numbers that should increment exit code

16.11.2 Output

`-env_push` push the current option environment
`-env_pop` pop the current option environment
`-env_save(Name)` save the current option environment
`-env_restore(Name)` restore the option environment to a previously saved one

16.11.3 New Flag Options

`fbl` dependent base class lookup in templates (default OFF).
`fcc` capitalize message categories (default OFF).
`fce` continue on `#error` (default OFF).
`fcs` continue on static assertion failure (default OFF).
`fdg` expansion of digraphs (default ON).
`fdm` comma from macro expansion does not delimit macro args (default OFF).
`fdt` delayed template parsing (default OFF).
`fee` expand environment variables (default ON).
`fei` underlying type for Enum is always Int (default OFF).
`fes` search enclosing scopes for friend tag `decls` (default OFF).
`ffv` implicit function to void pointer conversion (default OFF).
`ffw` allow friend `decl` to act as forward `decl` (default OFF).
`fgi` inline treated as GNU inline (default OFF).
`fho` header include guard optimization (default OFF).
`fla` locations for all diagnostics (default ON).
`flp` lax null pointer constants (default OFF).
`fma` microsoft inline asm blocks (default OFF).
`fms` microsoft semantics (default OFF).
`fon` support for C++ operator name keywords (default ON).
`frc` remove commas before `__VA_ARGS__` (default OFF).
`frd` redefine default params for class template function members (default OFF).
`fse` use smallest underlying type for enums (default OFF).
`fsi` search `#include` stack (default OFF).
`fso` return semantics override deduced return values (default OFF).
`fum` user declared move deletes only corresponding copy (default OFF).
`fur` allow unions to contain reference members (default OFF).

17 Revision History

17.1 Version 1.2

17.1.1 Highlights

Version 1.2 contains over 100 improvements to PC-lint Plus which are detailed in the following sections. Below is a summary of the most notable changes in this release.

- **New messages:** `473`, `621`, `1757`, `2452`, `2453`, `2454`, `2455`, `2456`, `2650`, `2662`, `2716`, `3707`, `9015`, `9069`, `9295`.
- **New options:** `-idlen`, `-misra_interpret`, `-format_category`.
- **New flag options:** `fcw`, `fgl`, `fmt`, `fmx`, `fzd`.
- **Language Support:** C++17 is now 100% supported including `constexpr` lambdas and structured bindings.
- **MISRA C 2012:** Added support for Rules 5.1, 5.2, 5.4, 9.2, 17.5, and 20.12. Improved support for Rules 11.7 and 11.9.
- **MISRA C++:** Added support for Rule 15-3-6. Improved support for Rules 5-2-7, 6-4-3, 6-4-6, and underlying type balancing rules.
- **Embedded:** Added support for target platforms using 16-bit and 32-bit bytes.
- **General Improvements:** Added supplemental messages to many of the errors in the 4xxx and 5xxx range, added warnings for incorrect use of more options, and about 50 other improvements.
- **Documentation:** Various improvements including new sections documenting system requirements and detailing how message suppressions are applied.
- **Defects Addressed:** Corrected over 30 defects.

17.1.2 Summary

17.1.2.1 Bugs Fixed

PCLP-1904	False positive 9007
PCLP-1957	False positive 915/917 for enumeration constants in C mode
PCLP-1997	False positive 9168 for static array member defined outside of class
PCLP-2026	Non-library regions incorrectly treated as library
PCLP-2099	Constrain ranges of inferenced values in Value Tracking messages
PCLP-2120	Support the 'l' strong type flag
PCLP-2158	False positive 1539 for template
PCLP-2183	Improved recognition of conditional variable modification for Value Tracking
PCLP-2190	Respect softeners for pointer differences when assigning or joining strong types
PCLP-2192	Result of bitwise OR not always calculated correctly
PCLP-2196	False positive 5731 for `__thread` in template
PCLP-2219	Improvements to handling of list initialization
PCLP-2226	False positive 527 for conditional break in switch case
PCLP-2229	False positive 438 for variable used in implicit construction of `std::initializer_list`
PCLP-2238	False positive 733 when assigning to pointer parameter in specific walk
PCLP-2241	False positive 9042 and 9082 for switch statements with leading `default` case
PCLP-2246	Message 967 not suppressed for `-header` files using library suppressions
PCLP-2249	Errors issued for uninstantiated templates
PCLP-2258	False positive 958 for structure with union members
PCLP-2261	False negative 737 for equality/relational tests
PCLP-2262	False positive 9090 when throwing a temporary object
PCLP-2268	False positive 587 for expressions with bitwise operations on types with different size or signedness
PCLP-2271	False positive 2434 for specific circumstances involving delete and return
PCLP-2278	Change 1997 to 1998 in set of permitted C++ years for `-std`
PCLP-2279	False positive 1529 for potentially unresolved overloaded operator&
PCLP-2289	Message 491 incorrectly issued for invalid name in macro definition
PCLP-2315	Crash when using `-h2` with empty caret indicator
PCLP-2366	Improved recognition of side effects from std::initializer_list initialization
PCLP-2400	Message 611 no longer issued for implicit conversions
PCLP-2417	Only emit "Resuming file" verbosity messages with -v<integer>
PCLP-2431	Correct handling of `-efunc` and `+efunc` options
PCLP-2445	False positive 641 for parenthesized enumeration constant in C mode
PCLP-2447	False positive 743 (negative character constant) in template instantiation

17.1.2.2 MISRA C 2012 Improvements

PCLP-1059	Support for MISR C 2012 Rule 20.12
PCLP-1061	Support for MISRA C 2012 Rule 9.2
PCLP-1066	Support for MISRA C 2012 Rule 17.5
PCLP-2176	Improved support for MIRSA C 2012 Rule 11.7
PCLP-2443	Improved support for MISRA C 2012 Rule 11.9
PCLP-2455	Support for MISRA C 2012 Rule 5.1
PCLP-2456	Support for MISRA C 2012 Rule 5.2
PCLP-2457	Support for MISRA C 2012 Rule 5.4

17.1.2.3 MISRA C++ Improvements

PCLP-545	Added support for MISRA C++ Rule 15-3-6
PCLP-1953	Implement MISRA's amended wording for balancing binary operators in MISRA C++
PCLP-2263	Improved support for MISRA C++ Rule 6-4-3
PCLP-2266	Improved support for MISRA C++ Rule 6-4-6
PCLP-2454	Void pointers no longer reported for Rule 5-2-7

17.1.2.4 General Improvements

PCLP-41	Support for 16-bit and 32-bit bytes
PCLP-1659	Extended exemptions for message 785
PCLP-1761	Consider base class fields for message 1401
PCLP-1875	Error 4374 suppressed for Visual Studio configurations
PCLP-1885	Tracking of multiple initialization variables in a `for` statement
PCLP-1887	Message 571 no longer issued for enumeration types
PCLP-1895	Soften 1938 for static-local and const-initialized variables
PCLP-1948	Better recognition of lint comments
PCLP-2119	Extend the strong type 'z' softener to casts of null pointer constants
PCLP-2133	Honor the value of the `fcc` flag option for summary output
PCLP-2147	Improved Value Tracking inferencing for booleans
PCLP-2160	`-fiz` no longer affects initialization of booleans
PCLP-2180	Issue 716 and not 774 for `while (1)` and `while (true)`
PCLP-2181	Only issue 1768 once per function
PCLP-2182	Extend value tracking depth for constexpr functions
PCLP-2189	Improved diagnostics for misuse of `-a` and `-s` options
PCLP-2218	Suppress message 948 for `if constexpr` conditions
PCLP-2233	Don't issue 587, 685, or 837 in instantiations
PCLP-2237	New warnings for improper use of `-i`
PCLP-2252	Support `-d`/`-u` options within files included via `-indirect` and improve behavior of combining `-env_push`/`-env_pop`, `-env_save`/`-env_restore`, and `-d`/`-u` options
PCLP-2253	Supplemental messages for compiler errors
PCLP-2254	Improved diagnostics for misuse of `-strong` boolean options
PCLP-2260	Unhelpful 746 when calling built-in atomic intrinsics in dependent contexts
PCLP-2265	Only issue message 9139 once per switch
PCLP-2273	Documentation improvements for VS2017 `pclp_config` configuration
PCLP-2302	Added symbol parameter to 9018
PCLP-2304	Consider tags used in `__builtin_offsetof` to be referenced
PCLP-2307	Support testing for non-null before deleting pointer
PCLP-2311	Improved validation of the `+group` option
PCLP-2312	Improved location information for message 9049
PCLP-2328	Improved error handling for `pclp_config`
PCLP-2339	Add operator argument to message 514
PCLP-2346	Message 857 softened for casts
PCLP-2388	Recognize `std::addressof` for 1529
PCLP-2398	Suppressing 893 with `-estring`
PCLP-2401	Suppress message 1506 in `final` classes
PCLP-2412	Improved value tracking inferencing
PCLP-2413	Name shadowing involving enumeration constants now reported by 578
PCLP-2416	Increased scope of message 445
PCLP-2420	False positive 9107 for member function template instantiated in a module
PCLP-2438	Message 1773 now issued for references
PCLP-2446	Issue 9045 messages in a deterministic order
PCLP-2469	Message 750 no longer reported when used in short-circuited `defined` operator
PCLP-2478	Support for response files introduced by compiler-specific option

17.1.2.5 New Features

PCLP-1052	Added `-idlen` and message 621
PCLP-1128	New Precision and Pre-determined Predicate Implementations
PCLP-1853	Message 1757 added
PCLP-1994	New `-misra_interpret` option to apply alternative interpretations to MISRA Rules
PCLP-2177	New `fcw` flag option to control whether 438 considers writes from called functions
PCLP-2239	New diagnostic for tentative array definition without a size in C mode
PCLP-2267	Improved C++17 Support
PCLP-2275	New `fgl` flag option to control the use of GNU line markers in preprocessed output
PCLP-2280	New `fmt` flag option to enable matching of template template-arguments to compatible templates

PCLP-2283	New `fmx` flag allows disabling of C++ member access control
PCLP-2284	New `fzd` flag to enable C++14 sized deallocation
PCLP-2357	New `-format_category` option to configure category representation
PCLP-2387	Add `\e` escape sequence for inserting ASCII escape into format strings
PCLP-2428	New message 2662 reports out of bounds pointer from scalar pointer
PCLP-2508	New message 2650 reports when a constant is out of range for part of a compound comparison operator

17.1.2.6 Documentation Enhancements

PCLP-2272	Update entry for messages 413 and 613 to include symbol
PCLP-2282	Better explain how lint comments in macro definitions are handled
PCLP-2301	Add section about using backslash escapes in options
PCLP-2396	Add version information to option and flag option tables
PCLP-2429	Add section documenting minimum and recommended OS/hardware requirements
PCLP-2430	Add section describing how message suppression options are applied
PCLP-2439	Correct notes about +emacro, +elibmacro, and +elibsym
PCLP-2444	Add note about new -efile behavior to "What's new" section

17.1.3 New Features

PCLP-1052	**Added `-idlen` and message 621** The `-idlen` option can be used to specify the number of significant characters in different types of C identifiers as well as whether identifiers are case-insensitive. Message `621` will report on clashes between identifiers that are not distinct within the initial significant characters.
PCLP-1853	**Message 1757 added** Message `1757` (discarded overriden post-increment/decrement), a PC-lint 9 message, is now supported by PC-lint Plus. Note that in PC-lint 9 this message was issued for both classes and scalars but in PC-lint Plus this message is issued only for classes; discard post-increment/decrement of scalar types is already reported by PC-lint Plus via message `2902`.
PCLP-1994 PCLP-2402	**New `-misra_interpret` option to apply alternative interpretations to MISRA Rules** The new options `-misra_interpret` and `+misra_interpret` can now be used to alter the default behavior of various MISRA guidelines. See the description of these options for usage details.
PCLP-2177	**New `fcw` flag option to control whether 438 considers writes from called functions** By default, PC-lint Plus will recognize writes of variables by called functions (such as when a variable is passed by reference and modified in the called function) and issue `438` (last value assigned not used) if the modified variable is not subsequently used. In some cases this behavior is not desired. In such cases, the `fcw` flag can be turned OFF so that writes by called functions are not considered for the purpose of message 438.
PCLP-2239	**New diagnostic for tentative array definition without a size in C mode** New message `2716` will note when a tentative array definition is encountered in C mode that does not include a size.
PCLP-2267	**Improved C++17 Support** PC-lint Plus now fully supports C++17. The C++17 features that were not previously supported include: • Exception specifications as part of the type system • `constexpr` lambdas • Template argument deduction of class templates • Using `auto` with non-type template parameters • Structured bindings • Pack expansions in `using` declarations

PCLP-2275 **New `fgl` flag option to control the use of GNU line markers in preprocessed output**
By default, PC-lint Plus generates GNU line markers instead of traditional `#line` directives when producing preprocessed output. GNU line markers have the form:

```
# line-number "filename" [flags]
```

If the `fgl` flag is turned OFF (it is ON by default), PC-lint Plus will instead generate `#line` directives in preprocessed output that have the form:

```
#line line-number ["filename"]
```

PCLP-2280 **New `fmt` flag option to enable matching of template template-arguments to compatible templates**
C++17 adds support for matching template template-arguments to compatible templates but the change as it exists in the published Standard is incomplete which can result in ambiguity errors for previously valid code. Because of this, the feature is not enabled by default and can be enabled by setting the `fmt` flag option to ON.

PCLP-2283 **New `fmx` flag allows disabling of C++ member access control**
The `fmx` flag (default ON) controls the enforcement of C++ member access control. If this flag is turned OFF, member access control will be disabled causing all protected and private access specifiers to be ignored and all classes to have public access.

PCLP-2284 **New `fzd` flag to enable C++14 sized deallocation**
The C++14 sized deallocation feature is disabled by default, setting the `fzd` flag to ON will enable it.

PCLP-1128 PCLP-1282 PCLP-1742 PCLP-1852 PCLP-1941 PCLP-2224 PCLP-2240 PCLP-2318 PCLP-2320 PCLP-2321 PCLP-2358

New Precision and Pre-determined Predicate Implementations

The Precision and Predetermined-predicate features have been rewritten correcting several bugs and improving corresponding diagnostics. The affected messages include 572 (excessive shift value), 587 (predicate can be pre-determined), 650 (constant out of range for operator), 685 (relational operator always evaluates to ...), and 734 (loss of precision). A new supplemental message provides information about the bit patterns involved when PC-lint Plus thinks this information is useful about the values involved. For example:

```
void foo(unsigned u) {
    if( (u & 0xCF) == 0x12 ) { }
}
```

will now elicit:

```
warning 587: predicate '==' can be pre-determined
    and always evaluates to false
    if( (u & 0xCF) == 0x12 ) { }
        ~~~~~~~~~~ ^  ~~~~

supplemental 891: incompatible bit patterns:
    U32_00000000000000000000000000010010 vs
    U32_000000000000000000000000??00????
    if( (u & 0xCF) == 0x12 ) { }
                   ^
```

to show that the bit pattern that corresponds to the constant expression can never match the known bits of the non-constant expression. See Precision, Viable Bit Patterns, and Representable Values for additional information.

PCLP-2357

New `-format_category` option to configure category representation

The new `-format_category` option can be used to change the way that message category names are presented in messages. See the description of `-format_category` for details.

PCLP-2387

Add `\e` escape sequence for inserting ASCII escape into format strings

The new `\e` escape sequence may be used inside the format string of the formatting options (`-format`, `-format_summary`, etc). This is useful when making use of ANSI escape sequences in format strings.

PCLP-2428

New message 2662 reports out of bounds pointer from scalar pointer

The new message 2662 will report when an out of bounds pointer is created offset from a pointer to a single value (typically the result of taking an object's address, as opposed to a pointer to an array or dynamically allocated buffer).

PCLP-2508

New message 2650 reports when a constant is out of range for part of a compound comparison operator

The new message 2650 will report when the '<' or '>' component of a '<=' or '>=' compound comparison operator is out of range for the provided type and constant expression.

17.1.4 Bugs Fixed

PCLP-1904 **False positive 9007**
A side-effect localized to a called function would sometimes be treated as a side-effect in the calling function which could result in false positive 9007 (side effects on right hand of logical operator) messages when the call appears on the RHS of a logical operator. This issue has been corrected.

PCLP-1957 **False positive 915/917 for enumeration constants in C mode**
When processing C source modules, PC-lint Plus would sometimes incorrectly issue message 915 or 917 (implicit conversion) when an enumeration constant was assigned to a variable with the same enumeration type. This issue has been corrected.

PCLP-1997 **False positive 9168 for static array member defined outside of class**
Message 9168 (variable has type alias name difference with previous declaration) was issued for static array member definitions outside the class when the array type involved a structure or typedef. For example:

```
//lint -w1 +e9168

class X {
    typedef int INT;
    static INT int_array[2];
};

X::INT X::int_array[2];
```

would result in the unintentional:

```
8  note 9168: variable 'X::int_array' has type alias name
    difference with previous declaration ('INT [2]' vs 'X::INT [2]')
X::INT X::int_array[2];
^
5  supplemental 891: previous declaration is here
    static INT int_array[2];
           ^
```

This issue has been corrected.

PCLP-2026 **Non-library regions incorrectly treated as library**
If a library header contained a lint comment, PC-lint Plus would sometimes treat the non-library region that follows as library for suppression purposes. This issue has been corrected.

PCLP-2099 **Constrain ranges of inferenced values in Value Tracking messages**
Value Tracking messages that included a range of values for a particular object would sometimes print a range whose bounds exceeded the values that could actually be stored by the source type. The Value Tracking information included in such messages will now represent the correct value range.

PCLP-2120 **Support the 'l' strong type flag**
Previously the 'l' flag of the -strong option was not honored. This issue has been corrected and assignment from library function return values will be treated as compatible for strong types that use this flag.

PCLP-2158 **False positive 1539 for template**
Message 1539 (member not assigned by assignment operator) was sometimes incorrectly issued within a template. This message is no longer issued in dependent contexts.

PCLP-2183 **Improved recognition of conditional variable modification for Value Tracking**
PC-lint Plus would not always recognize that a variable might have been modified when the modification occurred in a ternary operator for which PC-lint couldn't determine which branch was taken. This could result in false positive messages stemming from the assumption that the value of the variable had not changed. PC-lint Plus will no longer make assumptions about the value of the variable after the ternary operator in such situations.

PCLP-2190
PCLP-2243 **Respect softeners for pointer differences when assigning or joining strong types**
The pointer difference softeners for the `-strong` option were not being respected for assignment and join operations which could result in false positive 636 (strong type difference) messages. These softeners will now be honored.

PCLP-2192 **Result of bitwise OR not always calculated correctly**
In certain cases, the result of bitwise OR (`|`) was not correctly calculated during Value Tracking which could lead to false positive messages based on the assumed value. This issue has been corrected.

PCLP-2196 **False positive 5731 for `__thread` in template**
Error `5731` (initializer for thread-local variable must be a constant expression) was previously being issued when the `__thread` keyword was used in the declaration of a variable in a dependent context initialized with a non-type template parameter. This is allowed by GCC and the message will no longer be issued in dependent contexts.

PCLP-2219
PCLP-2360
PCLP-2361
PCLP-2362
PCLP-2365 **Improvements to handling of list initialization**
Variables inside of a braced initializer were not always being considered for purposes of access tracking, side effects, and reference binding. For example:

```
int foo(void) {
    int c = 0;
    int &n{c};
    return n;
}
```

would result in the undeserved:

```
warning 550: local variable 'c' declared in 'foo' not subsequently accessed
    int c = 0;
        ^
```

This change also corrects a potential false positive message 1762 (member function could be made const), and false negative 1570 (binding reference field to non-reference) and the unexpected results stemming from unrecognized side effects appearing in a braced initializer.

PCLP-2226 **False positive 527 for conditional break in switch case**
Message 527 (statement is unreachable due to unconditional transfer of control) was sometimes issued when a switch that otherwise returns on every path contains a conditional `break` statement. This issue has been corrected.

PCLP-2229 **False positive 438 for variable used in implicit construction of `std::initializer_list`**
The access of a variable's value inside of an implicit `std::initializer_list` was not being considered a "read" for read-write analysis which could lead to false positive 438 (last value assigned not used) messages. For example:

```
#include <vector>

int main() {
    int i = 10;
    std::vector<int> v;
    v = {i, i};
    return v[0];
}
```

Would previously result in 438 being issued for `i` which wasn't recognized as being read in the `std::initializer_list` assignment of `v`. This issue has been corrected.

PCLP-2238 **False positive 733 when assigning to pointer parameter in specific walk**
Message 733 (likely assigning address of local to outer scope pointer) was incorrectly issued when assigning the address of a local variable to a (non-reference) pointer parameter inside the function. This issue has been corrected.

PCLP-2241 **False positive 9042 and 9082 for switch statements with leading `default` case**
Messages 9042 (departure from MISRA switch syntax) and 9082 (switch should begin or end with default) were sometimes incorrectly issued when a swtich statement that started with a `default` statement was combined with other `case` statements. For example:

```
//lint -w1 +e9042 +e9082
void f(int i) {
    switch (i) {
    default:
    case 0:
        break;
    case 1:
        break;
    }
}
```

would incorrectly issue both messages. This issue has been resolved.

PCLP-2246 **Message 967 not suppressed for `-header` files using library suppressions**
Message 967 (header file does not have a standard include guard) was not suppressible using library suppressions (`-elib`/`-wlib`) for a header that was both used with a `-header` option and marked as a library function with with `+libh`. This issue has been corrected.

PCLP-2249 **Errors issued for uninstantiated templates**
Previously, PC-lint Plus would issue error messages for uninstantiated templates that would only be appropriate if the template was instantiated. PC-lint will no longer issue these errors for uninstantiated templates.

PCLP-2258 **False positive 958 for structure with union members**
Message 958 (padding needed for structure member) was previously incorrectly being issued in some cases when a structure contained a union member. This issue has been resolved.

PCLP-2261 **False negative 737 for equality/relational tests**
Message 737 (loss of sign in promotion) was previously not being issued for relevant equality and relational binary operators. This issue has been corrected.

PCLP-2262 **False positive 9090 when throwing a temporary object**
Message 9090 (switch case lacks unconditional break or throw) was previously incorrectly issued when a case statement was terminated with an unconditional `throw` of a temporary object. For example:

```
//lint -w1 +e9090
struct Z { };
void foo(int x) {
    switch (x) {
        case 1:
            break;
        case 2:
            throw Z();
        default:
            break;
    }
}
```

would elicit:

```
7  note 9090: switch case lacks unconditional break or throw
        case 2:
        ^
```

This issue has been corrected.

PCLP-2268 **False positive 587 for expressions with bitwise operations on types with different size or signedness**
In some cases, PC-lint Plus would incorrectly issue message 587 (predicate can be pre-determined) or terminate with an internal error when the result of an expression that involved bitwise operations on types with different size or signedness was used as a predicate. This issue has been resolved.

PCLP-2271 **False positive 2434 for specific circumstances involving delete and return**
In some very specific circumstances, a false positive 2434 (memory was potentially deallocated) could be issued when such deallocation was always followed by a return statement. This issue has been corrected.

PCLP-2278 **Change 1997 to 1998 in set of permitted C++ years for -std**
The `-std` option was previously accepting values of `c++97` and `c++1997` and rejecting values of `c++98` and `c++1998`. PC-lint Plus now accepts values indicating a year of 1998 and not 1997.

PCLP-2279 **False positive 1529 for potentially unresolved overloaded operator&**
Message 1529 (assignment operator should check for self-assignment) was previously incorrectly issued when an address of operator was used in the check for self-assignment but an unresolved overloaded operator function template existed for the same type.

PCLP-2289 **Message 491 incorrectly issued for invalid name in macro definition**
Message 491 (non-standard use of 'defined' preprocessor operator) was incorrectly issued when `#define` was used to define a macro with an invalid name (such as one starting with a number). This issue has been corrected.

PCLP-2315 **Crash when using -h2 with empty caret indicator**
PC-lint Plus could crash when using the message height option `-h2` without defining the caret indicator character to a non-space character. This issue has been corrected.

PCLP-2366 **Improved recognition of side effects from std::initializer_list initialization**
PC-lint Plus sometimes didn't recognize side effects that occured inside of a `std::initializer_list` which could result in false positive 1762 (member function could be made const) messages being issued. This issue has been corrected.

PCLP-2400 **Message 611 no longer issued for implicit conversions**
PC-lint Plus would previously issue 611 (cast between pointer to function type and pointer to object type) for both casts and implicit conversions. 611 is now only issued for casts. A separate warning will be issued for implicit conversions.

PCLP-2417 **Only emit "Resuming file" verbosity messages with -v<integer>**
According to the Reference Manual, "Resuming file" verbosity messages are only emitted when using the option `-v<integer>` but such messages were also being incorrectly issued when using `-vf`. This issue has been corrected.

PCLP-2431
PCLP-2432 **Correct handling of -efunc and +efunc options**
Previously, `+efunc` options were not honored and a `-efunc` option would unconditionally suppress the corresponding message without taking into consideration other parameterized options that might want to enable the message. Both of these issues have been corrected.

PCLP-2445 **False positive 641 for parenthesized enumeration constant in C mode**
Parenthesized enumeration constants were not always recognized as having the same type of its corresponding enumeration in C mode which could result in false positive 641 (implicit conversion of enum to integral type) messages. This issue has been corrected.

PCLP-2447 **False positive 743 (negative character constant) in template instantiation**
Message 743 (negative character constant) was sometimes incorrectly reported during the instantiation of a template with a non-type template parameter of character type even though the non-type template argument was not written as a character constant. Message 743 will no longer be issued for template arguments.

17.1.5 MISRA C 2012 Improvements

PCLP-1059	**Support for MISR C 2012 Rule 20.12** PC-lint Plus now supports MISRA C 2012 Rule 20.12 via the new message 9015 (macro parameter used with and without '#/##' subject to further replacement).
PCLP-1061	**Support for MISRA C 2012 Rule 9.2** PC-lint Plus now supports MISRA C 2012 Rule 9.2 via the new message 9069 (initializer needs braces or designator).
PCLP-1066	**Support for MISRA C 2012 Rule 17.5** MISRA C 2012 Rule 17.5 is now supported via the new message 473 (argument is of insufficient length for sized array parameter).
PCLP-2176	**Improved support for MIRSA C 2012 Rule 11.7** PC-lint Plus now issues message 9295 for the non-error violations of MISRA C 2012 Rule 11.7 that were not previously diagnosed.
PCLP-2443	**Improved support for MISRA C 2012 Rule 11.9** Message 9080 (integer null pointer constant is not the NULL macro), which supports MISRA C 2012 Rule 11.9, is no longer issued when initializing arrays of pointers with `{0}` as clarified by MISRA.
PCLP-2455	**Support for MISRA C 2012 Rule 5.1** PC-lint Plus now supports MISRA C 2012 Rule 5.1. See the updated `au-misra3.lnt` file for details.
PCLP-2456	**Support for MISRA C 2012 Rule 5.2** PC-lint Plus now supports MISRA C 2012 Rule 5.2. See the updated `au-misra3.lnt` file for details.
PCLP-2457	**Support for MISRA C 2012 Rule 5.4** PC-lint Plus now supports MISRA C 2012 Rule 5.4. See the updated `au-misra3.lnt` file for details.

17.1.6 MISRA C++ Improvements

PCLP-545	**Added support for MISRA C++ Rule 15-3-6** MISRA C++ Rule 15-3-6 is now supported via message `1775` (catch block does not catch any declared exceptions). Additionally, message 1775 has been updated to include a supplemental message with the location of the catch block that will pre-empt the catch block being diagnosed.
PCLP-1953	**Implement MISRA's amended wording for balancing binary operators in MISRA C++** MISRA recently clarified rules for balancing underlying types for integral operands. PC-lint Plus implements those clarifications.
PCLP-2263	**Improved support for MISRA C++ Rule 6-4-3** PC-lint Plus no longer issues message `9042` (departure from MISRA switch syntax) when a `case` statement is terminated by an unconditional `throw` expression.
PCLP-2266	**Improved support for MISRA C++ Rule 6-4-6** There was an edge case for MISRA C++ Rule 6-4-6 that didn't have a message mapped by the au-misra-cpp.lnt file. In cases where a `switch` condition had `enum` type but not all the enumerators were represented in the `switch` cases, there was no message being issued with MISRA violation verbiage. Message `787` reports this specific case and was added to au-misra-cpp.lnt.
PCLP-2454	**Void pointers no longer reported for Rule 5-2-7** MISRA have clarified that conversions involving void pointers are not intended to be covered by this Rule and such conversions will no longer be reported by message `9176` (pointer type converted to unrelated pointer type).

17.1.7 General Improvements

PCLP-41	**Support for 16-bit and 32-bit bytes** PC-lint Plus now supports 16-bit and 32-bit bytes used by some embedded architectures. The number of bits in a byte can be set with the `-sb` option, e.g. `-sb16` or `-sb32`. See the description of the `-s` and `-sb` options for additional details.
PCLP-1659	**Extended exemptions for message 785** Message 785 (too few initializers for aggregate) has always provided an exception for using `{0}` which is instead reported by Elective Note 943. This exception has been expanded such that using the initializers `{}` and `{nullptr}` will now be reported by 943 instead of 785 as well.
PCLP-1761	**Consider base class fields for message 1401** Message 1401 (non-static data member not initialized by constructor) now reports when base class fields are not initialized as a result of running the target constructor. By default, PC-lint Plus will report on fields up to two base class levels deep, this can be increased by increasing the value of `-vt_depth`.
PCLP-1875	**Error 4374 suppressed for Visual Studio configurations** Message 4374 (virtual function has different calling convention attributes than the function it overrides) is now suppressed by Visual Studio configurations genereated with `pclp_config` as Visual Studio allows this in some cases.
PCLP-1885	**Tracking of multiple initialization variables in a `for` statement** Messages 443 (for statement initializer variable is inconsistent with modification variable) and 445 (reuse of for loop variable) now better handle multiple initialization variables. Previously PC-lint Plus would select one variable as the initialization variable which could result in false positives when a different variable was modified.
PCLP-1887	**Message 571 no longer issued for enumeration types** Message 571 (cast results in sign extension) is no longer issued when the conversion involves an enumeration type which matches the behavior of PC-lint 9.
PCLP-1895	**Soften 1938 for static-local and const-initialized variables** Message 1938 (constructor accesses global data) will no longer be issued for static local variables or static variables with a constant initializer.
PCLP-1948	**Better recognition of lint comments** Previously, PC-Lint Plus would treat any comment that starts with "lint" as a lint comment even if there is not whitespace between the "lint" and what comes after, e.g.: `//lintq-message=ABC` was treated as a valid lint comment with the character coming after "lint" being ignored regardless of its value. The new handling will only consider a comment to be a lint comment if it either 1) starts with "lint" followed by a whitespace character or 2) the entire body of the comment consists of "lint" (which will be reported by warning 2439: lint comment does not contain any options).
PCLP-2119	**Extend the strong type 'z' softener to casts of null pointer constants** The 'z' strong type softener (see the `-strong` option) will now suppress strong type difference messages when a strong pointer type (or pointer to strong type) is combined with a null pointer constant that is cast to a pointer type.

PCLP-2133 **Honor the value of the fcc flag option for summary output**
The message categories displayed in summary output (-summary) will now be subject to the value of the fcc flag controlling capitalization of message categories.

PCLP-2147 **Improved Value Tracking inferencing for booleans**
Inferencing of boolean types has been improved. For example:

```
void f(bool s) {
    if (s) {
        if (!s) { }
    }
}
```

will now issue message:

```
info 774: boolean condition for 'if' always evaluates to 'false'
        if (!s) {
        ^
supplemental 831: logical not yields 0
        if (!s) {
            ^~
supplemental 831: inference yields 1
    if (s) {
    ^   ~
```

PCLP-2160 **-fiz no longer affects initialization of booleans**
The fiz flag determines whether or not variables initialized to zero are considered to be "written" for the read-write analysis feature. Previously, when this flag was OFF, initializing a boolean with a zero was not treated as a write but initializing with `false` was, which was inconsistent. Zero is treated as a special value since it is common to initialize integer and pointer variables to this value. Boolean types only have two values and it is not clear that `false` is any more "special" than `true` for boolean initialization so boolean types are no longer considered for the purpose of this flag. Booleans are still respected by the -fiw flag option which doesn't consider any initializations as a write.

PCLP-2180 **Issue 716 and not 774 for while (1) and while (true)**
Message 716 (infinite loop via while) was previously issued for `while (1)` constructs but is now also issued for `while (true)`. Message 774 (boolean condition always evaluates to true/false) will no longer be issued in these two cases.

PCLP-2181 **Only issue 1768 once per function**
Message 1768 (differing accesses for overridden virtual function) was previously being issued for both the in-class declaration of a member function and an out-of-class definition. This message will now be issued only for the in-class declaration.

PCLP-2182 **Extend value tracking depth for constexpr functions**

The value tracking depth (controlled via the `-vt_depth` option) specifies the maximum call depth for which specific function calls are walked during value tracking. A sufficiently low value tracking depth may have unexpected results caused by PC-lint Plus not walking a function past the specified depth. For example:

```
1  template <typename T, int N>
2  constexpr unsigned func_elems(T (&)[N]) { return N; }
3
4  #define macro_elems(a) (unsigned)(sizeof(a)/sizeof(a[0]))
5
6  struct X {
7      unsigned char buffer[100]{};
8
9      int char_at1(unsigned idx) {
10         return idx < macro_elems(buffer) ? buffer[idx] : -1;
11     }
12
13     int char_at2(unsigned idx) {
14         return idx < func_elems(buffer) ? buffer[idx] : -1;
15     }
16 };
17
18 void foo() {
19     X x;
20     x.char_at1(500);
21     x.char_at2(500);
22 }
```

would previously result in the message:

```
14  warning 415: likely out of bounds pointer access: excess of 401 bytes
        return idx < func_elems(buffer) ? buffer[idx] : -1;
                                          ^      ~~~

21  supplemental 894: during specific walk char_at2(500)
    x.char_at2(500);
      ^
```

for the call to `char_at2` but not the functionally equivalent `char_at1`. The difference is that `char_at1` uses a macro to calculate the size of the array while `char_at2` uses a (`constexpr`) function call which isn't walked because doing so would exceed the default value tracking depth. Because of this, PC-lint Plus doesn't know what `idx` is being compared to and since the value in the call from `foo` exceeds the size of the array, a message is issued. To prevent surprising behavior such as this, `constexpr` functions will now be walked when the function call depth is equal to the max depth. A similar exemption already existed for literal operators. Of course increasing the value tracking depth using `-vt_depth` will also allow PC-lint Plus to perform a deeper analysis and mitigate false positives and false negatives caused by a horizon effect.

PCLP-2189 **Improved diagnostics for misuse of `-a` and `-s` options**
The `-a` and `-s` options used to specify alignment and sizes of fundamental types cannot be used inside of a module. Previously PC-lint Plus would issue a warning when such an option was included in a lint comment but no warning was issued when such an option was encountered in an indirect file that was included from inside the module. The warning will now be issued when these options are encountered in any way while processing a module.

PCLP-2218 **Suppress message 948 for `if constexpr` conditions**
Message 948 (operator always evaluates to true/false) will no longer be emitted for C++17 `if constexpr` conditions. For example, this code:

```
int foo(void) {
    if constexpr(sizeof(wchar_t) == 2) { }
    else { return 1; }
}
```

will no longer generate message 948.

PCLP-2233 **Don't issue 587, 685, or 837 in instantiations**
Messages 587 (predicate can be pre-determined), 685 (relational operator always evaluates to ...), and 837 (switch condition is a constant expression) are no longer issued inside of functions instantiated from templates. These messages are instead issued once for the template in which they occur, previously they were issued inside the template as well as within every function instantiated from the template.

PCLP-2237 **New warnings for improper use of `-i`**
PCLP-2334 PC-lint Plus will now issue a warning when the `-i` option is provided the absolute path of 1) a directory that does not exist or is not accessible or 2) a file. A not uncommon mistake is to write e.g. `-i/a/b/c/test.c` instead of `-i/a/b/c/ test.c`, the former not accomplishing anything useful. A warning message will now be issued in such cases. A warning will also be issued for paths that start with a drive letter on non-Windows platforms and paths that begin with a tilde (~) which was likely intended to be expanded to a home directory (PC-lint Plus does not expand tildes).

PCLP-2252 **Support -d/-u options within files included via -indirect and improve behavior of combining -env_push/-env_pop, -env_save/-env_restore, and -d/-u options**
Previously, `-d/-u` options were not respected when appearing inside of a configuration file that was incuded via a `-indirect` option inside of a module. Such options will now be honored. Additionally, when empoying certain combinations of `-env_push/-env_pop`, `-env_save/-env_restore`, and `-d/-u` options, the intended behavior was not always being realized. In particular, when a `-d/-u` option appeared after a `-env_push` option, if a `-env_save` option was used to save the option environment and was then followed by a corresponding `-env_pop` option before the saved state was restored via a `-env_restore`, the `-d/-u` option would not be in effect after the `-env_restore`. For example, the options:

```
-dX=15
-env_push
-dX=10
-env_save("test")
-env_pop
-env_restore("test")
```

should result in `X` having the value `10` after the `-env_restore("test")` option is processed but was previously expanding to `15` instead. A similar issue affected parameterized suppression options in the same way that `-d/-u` options were affected. These issues have been resolved.

PCLP-2253 **Supplemental messages for compiler errors**
Compiler error messages (those in the ranges 4xxx and 5xxx) were not generally accompanied by supplemental messages. Many of these messages are now issued with supplemental messages which provide additional detail about the circumstances of the error.

PCLP-2254 **Improved diagnostics for misuse of -strong boolean options**
PC-lint Plus will now warn when using a `-strong` option that attempts to set more than one strong boolean type.

PCLP-2260 **Unhelpful 746 when calling built-in atomic intrinsics in dependent contexts**
PC-lint Plus supports a variety of built-in functions and intrinsics understood by multiple compilers. In some situations, PC-lint Plus was issuing message `746` (call not made in the presence of a prototype) when calling certain of these functions. Since the whole point of having built-in support for these functions is not having to provide a prototype to use them, this message is not useful. PC-lint Plus will no longer issue 746 for built-in functions.

PCLP-2265 **Only issue message 9139 once per switch**
Message `9139` (case label follows default in switch statement) was previously issued for every `case` statement that followed a `default` in a `swtich` statement. This message is now issued only for the first such `case` statement making suppressing easier and reducing unhelpful messages.

PCLP-2273 **Documentation improvements for VS2017 pclp_config configuration**
The description for the `vs2017` and `vs2017_64` `pclp_config` targets incorrectly referenced "Visual Studion 2015", this has been corrected. Additionally, a note has been added to section 2.2.4 of the Reference Manual differentiating between 32-bit and 64-bit Visual Studio configurations.

PCLP-2302 **Added symbol parameter to 9018**
Message `9018` (union declared) now includes a symbol parameter to allow suppression for specific union declarations using `-esym`.

PCLP-2304 **Consider tags used in `__builtin_offsetof` to be referenced**
Class and struct tags used with the `__builtin_offsetof` operator will no longer be reported as not referenced.

PCLP-2307 **Support testing for non-null before deleting pointer**
Message `1540` (non-static pointer data member not deallocated nor zeroed by destructor) was previously issued for a destructor that deleted a member pointer only after checking that the pointer was non-null. Such a check will no longer elicit this message.

PCLP-2311 **Improved validation of the `+group` option**
The `+group` option now correctly enforces the documented requirement that group names begin with a letter and no longer allow groups to be created with empty names.

PCLP-2312 **Improved location information for message 9049**
Message `9049` (increment/decrement operation combined with other operation with side-effects) was previously issued with a location representing the start of the statement making it difficult to suppress in certain cases (such as with `-emacro` when the increment/decrement operation was a result of macro expansion). The primary location presented is now the location of the increment/decrement operation and the other side-effect operations are now highlighted.

PCLP-2328 **Improved error handling for `pclp_config`**
`pclp_config` will now search both the current directory and the directory containing the `pclp_config` script for the `compilers.yaml` configuration file by default. Additional warnings have been added to handle cases where expected options are missing.

PCLP-2339 **Add operator argument to message 514**
Message `514` (boolean argument to bitwise/arithmetic operator) now includes the corresponding operator in the diagnostic.

PCLP-2346 **Message 857 softened for casts**
Message `857` (argument 1 is not compatible with argument 2 in call to function) now considers casts in the arguments to memcpy/memmove/memcmp which can be used to suppression this message. This change also reflects a clarification by MISRA on the correspond MISRA C3 Rule 21.15.

PCLP-2388 **Recognize `std::addressof` for 1529**
Message `1529` (assignment operator should check for self-assignment) now recognizes a self-assignment test using `std::addressof` instead of the address of operator (`&`).

PCLP-2398 **Suppressing 893 with `-estring`**
Message `893` (expanded from macro) now contains a string parameter that can be suppressed using `-estring(893,` *`name)`*.

PCLP-2401 **Suppress message 1506 in `final` classes**

Message 1506 (virtual function call in constructor/destructor) was previously suppressed when the call was qualified with the scope operator, the called function was marked final, or the class within which the called function was declared was marked final. The last condition has been changed to refer to the class of the constructor or destructor instead of the class in which the called function was declared to accommodate the case where the virtual function in question is only declared in a base class. For example:

```
class A {
public:
    virtual void f() { }
};
class B final : public A {
public:
    B() {
        f();
    }
};
```

will no longer elicit 1506. Additionally, prefixing a virtual function call with `this->` will no longer avoid the message. The criteria for the companion elective note 1933 have been duly updated as well.

PCLP-2412 **Improved value tracking inferencing**

Value Tracking will now correctly inference expressions where variables are on the right-hand side of a comparison operator, e.g. `5 > a` instead of `a < 5`.

PCLP-2413 **Name shadowing involving enumeration constants now reported by 578**

Message 578 (declaration hides identifier) now reports when a declaration hides the name of an enumeration constant or the declaration of an enumeration constant hides an identifier.

PCLP-2416 **Increased scope of message 445**

Message 445 (re-use of loop variable) previously only reported when the variable being initialized by the first clause in an outer `for` statement was re-used in a nested `for` loop. PC-lint Plus will now additionally consider variables modified in the second and third clauses of an outer `for` statement when there is no initialization variable and report re-uses of such variables.

PCLP-2420 **False positive 9107 for member function template instantiated in a module**
PCLP-2421

Message 9107 (header cannot be included in more than one translation unit because of the definition of symbol) was incorrectly being issued when a member function template declared in a header was instantiated as the result of instantiating another member function template in the same header resulting from an instantiation that originated in a module.

PCLP-2438 **Message 1773 now issued for references**

Message 1773 (casting away const/volatile) will now be issued when casting away const/volatile from a reference, e.g.:

```
void f(const int& x) {
    int& b = (int&)x;
}
```

PCLP-2446 **Issue 9045 messages in a deterministic order**
In the Windows build of PC-lint Plus, when multiple `9045` (complete definition of symbol is unnecessary in this translation unit) messages were emitted, the order of these messages was not always consistent between runs. Multiple 9045 messages will now be issued in a deterministic order on all platforms.

PCLP-2469 **Message 750 no longer reported when used in short-circuited `defined` operator**
Message `750` (local macro not referenced) was previously issued for a macro that appeared in a `defined` operator on the right-hand side of a short-circuited logical operator. For example, message 750 was previously issued for macro `B` in the example:

```
#define B
#if defined(A) && defined(B)
...
#endif
```

Message 750 will no longer be issued in such cases.

PCLP-2478 **Support for response files introduced by compiler-specific option**
The `imposter.c` program that ships with PC-lint Plus now supports response files (files that contain compiler invocation options) that are introduced with a compiler option. For example, the IAR compiler allows a response file to be specified with the `-f` option. The new `IMPOSTER_RSP_INTRO_ARG` environment variable can now be set to a string that represents this option (e.g. `"-f"`). When this environment variable is set and `imposter.c` encounters an option matching its value, the next argument is assumed to be a response file and the options it contains are logged instead of the option itself.

17.1.8 Documentation Enhancements

PCLP-2272	**Update entry for messages 413 and 613 to include symbol** Messages 413 (likely use of null pointer) and 613 (potential use of null pointer) are issued with a symbol when one is available. This symbol was not included in the text of the message in the documentation. This issue has been corrected.
PCLP-2282	**Better explain how lint comments in macro definitions are handled** A new section, 4.1.2 Lint Comments inside of Macro Definitions, has been added to clarify how lint comments in macro definitions are handled.
PCLP-2301	**Add section about using backslash escapes in options** The new subsection, 4.1.12 Escaping Special Characters, has been added to the section 4.1 Rules for Specifying Options that describes how the backslash character can be used to escape special characters inside of options when the `fbe` flag is ON.
PCLP-2396	**Add version information to option and flag option tables** The option and flag option summary tables in Chapter 4 of the Reference Manual now include the PC-lint Plus version in which the option was introduced.
PCLP-2429	**Add section documenting minimum and recommended OS/hardware requirements** A new section 2.1 System Requirements has been added detailing the Operating System and Hardware requirements for PC-lint Plus.
PCLP-2430	**Add section describing how message suppression options are applied** The new section 4.9 How Suppression Options are Applied describes in detail the message suppression process including how different suppression options work and interact with each other.
PCLP-2439	**Correct notes about +emacro, +elibmacro, and +elibsym** There was a note in the description of `+emacro` claiming that it only works to undo a previously seen `-emacro`. This was true in PC-lint 9.0 but is not true in PC-lint Plus and the note has been removed. A similar note was added to the descriptions of the `+elibmacro` and `+elibsym` options which do only work to undo a previously seen `-elibmacro` or `-elibsym` option (which was also the case in PC-lint 9.0).
PCLP-2444	**Add note about new -efile behavior to "What's new" section** A note about the change in behavior for `-efile` was added to 16 Differences from PC-lint 9.0.

17.2 Version 1.1

17.2.1 Summary

17.2.1.1 Bugs Fixed

PCLP-1454	Improved handling of `goto` for read/write analysis
PCLP-1483	Single line suppressions not always honored when location is macro name
PCLP-1512	Improved handling of `-emacro`
PCLP-1711	Don't create null inferences when comparing to the `this` pointer
PCLP-1826	False positive 9107 for member instantiations
PCLP-1841	Message 923 not issued for `reinterpret_cast`
PCLP-1894	False positive 641 for parenthesized enumeration constant in C mode
PCLP-1908	False positive 838 for multiple early returns
PCLP-1919	Improved recognition of volatile assignment as an impurity
PCLP-1909	False positive 705 for `%jd` conversion specifier with `intmax_t` defined as long long
PCLP-1910	Statements without side effects not diagnosed in specific circumstances
PCLP-1914	Message 826 no longer issued for `dynamic_cast`
PCLP-1920	Improved custodial semantics for reference parameters
PCLP-1921	Custodial semantic not properly applied for operator call expressions
PCLP-1942	False positive 568/775 for reference member variables
PCLP-1943	False positive 568 for reference types
PCLP-1945	Message 916 now only reports pointer-to-pointer conversions
PCLP-1955	Improperly suppressed messages from library headers
PCLP-1960	Messages 900 and 870 not issued when using 2 passes
PCLP-1974	Support for bidirectional pre-loop inference test direction
PCLP-1993	False positive 1924 for substituted non-type template parameters of enumeration type
PCLP-1995	Reset position indicator character when using `-h2`
PCLP-2001	Improved handling of pointer parameters for message 733
PCLP-2013	Message 2702 now issued regardless of suppression state of 528
PCLP-2014	False positive 449
PCLP-2015	False positive 9048 for use of enum constant non-type template parameter
PCLP-2029	Improved handling of angle brackets in message 773
PCLP-2068	Extraneous supplemental message
PCLP-2069	Don't analyze assembly statements
PCLP-2073	Crash when using allocation semantics in user-defined function semantics
PCLP-2077	Incorrect null inference for equality check against non-null pointer
PCLP-2078	Consistent diagnosis of null pointers for array indexing and pointer arithmetic
PCLP-2101	False positive 9176 for implicit conversion of this pointer to base class pointer
PCLP-2102	False Positive 743 for wide character constants
PCLP-2105	False positive 527 after switch containing conditional return
PCLP-2106	Improved Value Tracking of structures in C
PCLP-2108	False positive 1751 for macro defined in header that expands to anonymous namespace in main source file
PCLP-2125	Remove message 504 from MISRA author files
PCLP-2127	False negative 1506
PCLP-2128	Add POD semantics to Standard C library functions
PCLP-2140	False positive 1415 for pointers to void
PCLP-2151	False positive type alias differences for function template specializations
PCLP-2153	Internal error for improper user-defined function return allocation semantics
PCLP-2155	False positive 9034 involving compound assignment
PCLP-2156	Message 9003 no longer issued for local static variables
PCLP-2157	False positive 909 and 910 for certain casts
PCLP-2159	False positive 1727 for member function of class template specializations
PCLP-2163	Strong type of enumerator not recognized when typedef uses incomplete type
PCLP-2169	False positive 758 for implicit instantiations
PCLP-2179	Internal error related to message 9027 when issued in C++ code
PCLP-2191	Structure member initialization status not properly merged after being initialized in both branches of an if statement in C mode
PCLP-2203	Improved Value Tracking of unknown function parameters of structure type

PCLP-2208	Message 1415 issued for type-dependent arguments

17.2.1.2 MISRA C 2004 Improvements

PCLP-1874	Improved support for MISRA C 2004 Rule 13.1
PCLP-1882	Improved support for MISRA C 2004 Rules 11.1 and 11.2
PCLP-1929	Improved handling of MISRA C 2004 Rule 10.1
PCLP-2009	Improved support for MISRA C 2004 Rule 10.1
PCLP-2018	Improved support for MISRA C 2004 Rules 6.1 and 6.2
PCLP-2037	Improved support for MISRA C 2004 Rules 6.1/6.2 and MISRA C++ Rule 5-0-11

17.2.1.3 MISRA C 2012 Improvements

PCLP-1037	Improved support for MISRA C 2012 Rule 20.8
PCLP-1528	Update messages used for Rule 11.7 in au-misra3.lnt
PCLP-1776	Improved support for MISRA C 2012 Rule 10.3
PCLP-1778	Improved support for MISRA C 2012 Rule 11.4
PCLP-1779	Improved support for MISRA C 2012 Rule 11.4
PCLP-1780	Improved support for MISRA C 2012 Rule 11.4
PCLP-1781	Improved support for MISRA C 2012 Rule 11.5
PCLP-1879	Improved support for MISRA C 2012 Rule 10.1
PCLP-1880	Improved support for MISRA C 2012 Rule 10.1
PCLP-1881	Improved support for MISRA C 2012 Rule 10.2
PCLP-1898	Improved support for MISRA C 2012 Rule 10.7
PCLP-2017	Do not classify wide character constants as having essentially character type in MISRA C 2012
PCLP-2030	Improved support for MISRA C 2012 Rule 18.8
PCLP-2198	Incorrect MISRA essential type calculation for modulo operations
PCLP-2220	Improved support for MISRA C 2012 Rule 11.5

17.2.1.4 MISRA C++ Improvements

PCLP-1952	Improved support for MISRA C++ Rule 5-3-4
PCLP-1989	Improved support for MISRA C++ Rule 3-1-1
PCLP-2031	Corrections to au-misra-cpp.lnt for MISRA C++ Rules 17-0-1 and 17-0-2
PCLP-2117	Improved support for MISRA C++ Rule 7-3-6
PCLP-2150	False positive 9113 for compound assignment
PCLP-2170	Increased scope of "related types" for message 9176

17.2.1.5 General Improvements

PCLP-1768	Improved handling when all paths return in `if` statement
PCLP-1790	Improved handling of placement `new` for read-write analysis
PCLP-1795	Improved handling of mutable members in side effects determination
PCLP-1799	Improved handling of user-defined return value semantics
PCLP-1800	Improved handling of enumeration constants of strongly typed `enums`
PCLP-1813	The `+efreeze` option now always prevents single-line suppressions
PCLP-1899	Don't issue 1564 when integer literal is converted/cast
PCLP-1911	Assume side-effect when passing pointer offset to non-const pointer parameter
PCLP-1932	Prototype information now included in message 1411
PCLP-1934	"Could be const" messages softened for typedefs from macros
PCLP-1951	Assume modification of initialized arguments when using -fai
PCLP-1956	Clarified message text for message 956
PCLP-1965	Add symbol information to message 9103
PCLP-1967	Issue message 564 for volatile reads
PCLP-1969	Improved handling of deallocation tracking in `if` statements
PCLP-1970	False positive 773 for named casts
PCLP-1971	Exempting logical not from message 1564

PCLP-1976	Improved performance for next-statement suppressions
PCLP-1980	1938 no long issued for `constexpr` variable
PCLP-1985	Improved handling of structures initialized by functions with using `fai`
PCLP-1987	Improved output format for `enum` essential types in essential type messages
PCLP-1992	False positive 522 in `__asm` statement
PCLP-2002	Message 785 softened for aggregate initialization with extra braces
PCLP-2006	Recognition of aggregate initialization evaluation order in C++11
PCLP-2024	Support flexible array members in base classes
PCLP-2039	Clarifications to text of message 9128
PCLP-2050	Improved `pclp_config` support for macro definition that contain spaces and quotes in Visual Studio configurations
PCLP-2074	Improved Value Tracking handling of `while` statement condition variable declarations
PCLP-2075	Improve application of dynamic allocation semantics
PCLP-2081	Message 1793 no longer issued for functions with an rvalue reference qualifier
PCLP-2088	Improved handling of premature termination due to stack overflow
PCLP-2098	Improved Value Tracking for unconditional assignment in both branches of an `if` statement
PCLP-2152	Improved handling of friends and templates for 9004
PCLP-2161	Improved handling of misspelled parameterized suppression options
PCLP-2164	Add symbol parameter to null pointer dereference messages when available
PCLP-2166	Message 904 issued multiple times for repeated declarations
PCLP-2178	Suppress message 1788 for variables marked as unused
PCLP-2185	Improved error handling for invalid size and alignment options
PCLP-2195	Improved handling of possibly null information in user-defined return semantics
PCLP-2200	Improved handling of buffers used with placement `new`
PCLP-2205	Improved support for C++17 `constexpr if`
PCLP-2214	Added `env-html.lnt`, `env-html.js`, and `env-xml.lnt` files

17.2.1.6 New Features

PCLP-1531	Support for IAR compilers and IAR Workbench
PCLP-1889	New message 3450 - subtracting member from 'this' pointer
PCLP-2051	New `fbe` flag option and backslash escapes
PCLP-2054	Added support for MISRA C 2012 AMD-1 Rule 12.5
PCLP-2055	Added support for MISRA C 2012 AMD-1 Rule 21.13
PCLP-2057	Added support MISRA C 2012 AMD-1 Rule 21.15
PCLP-2058	Added new message (9098) to support MISRA C 2012 AMD-1 Rule 21.16
PCLP-2059	Add partial support for MISRA C 2012 AMD-1 Rule 21.17
PCLP-2083	Allow the C++17 fallthrough attribute to suppress messages 616 and 825
PCLP-2132	The `-help` option now responds to message numbers
PCLP-2201	New exit command added to Value Tracking debugger

17.2.1.7 Documentation Enhancements

PCLP-1926	Add "Flow of Execution" section (1.2) to Reference Manual
PCLP-1949	Fix runaway text in Reference Manaual
PCLP-1966	Update description of message 9049
PCLP-2008	Reference the `flf` flag in the Value Tracking section of the Reference Manual
PCLP-2012	Improvements to descriptions of select error messages
PCLP-2016	`fwc` flag removed from manual
PCLP-2049	Miscellaneous documentation corrections
PCLP-2172	Clarify that `-egrep` doesn't match text injected via `+typename`
PCLP-2210	Update description of message 916 to remove MISRA C++ support statement

17.2.2 Bugs Fixed

PCLP-1454 **Improved handling of goto for read/write analysis**
Undeserved 838 (previous value assigned not used) messages were previously sometimes issued for variables where a `goto` statement resulted in a skipped intermediate assignment. This has been corrected.

PCLP-1483 **Single line suppressions not always honored when location is macro name**
In some cases, when the primary location of a message points to a macro name, the single-line suppression syntax was not effective. For example:

```
/*lint -w1 +e2705 */
#define FTQ volatile

FTQ int foo(); //lint !e2705
```

would result in the message:

```
info 2705: type qualifier 'volatile' applied to return type has no effect
FTQ int foo(); //lint !e2705
^~~~
supplemental 893: expanded from macro 'FTQ'
#define FTQ volatile
            ^
```

This issue has been corrected.

PCLP-1512 **Improved handling of -emacro**
Previously, the `-emacro` option was not effective when the location of the message corresponded to the expansion of a macro argument within another macro. For example:

```
#define M1(a, b, c) (a b c)
#define M2(a, b, c) M1(a, b, c)

void foo(int *i) {
    *i = M2(1, +, 0);
}
```

causes PC-lint Plus to issue:

```
5  info 835: zero given as right argument to + in a constant expression
    *i = M2(1, +, 0);
               ^  ~
2  supplemental 893: expanded from macro 'M2'
#define M2(a, b, c) M1(a, b, c)
                          ^  ~
1  supplemental 893: expanded from macro 'M1'
#define M1(a, b, c) (a b c)
                       ^ ~
```

The option `-emacro(835, M1)` was previously not sufficient to suppress this message. Such suppressions will now work as expected.

PCLP-1711 **Don't create null inferences when comparing to the `this` pointer**
PC-lint was sometimes inferring that a pointer that compared against the `this` pointer could be null. For example:

```
//lint -w1 +e413
struct Foo {
    void Bar(Foo* pOther) {
        if(pOther != this) {
            pOther->Baz();
        }
    }
    void Baz();
};
```

Would elicit:

```
warning 413: likely use of null pointer
            pOther->Baz();
            ~~~~~~  ^
supplemental 831: inference yields nullptr
        if(pOther != this) {
        ^  ~~~~~~~~~~~~~~
```

This issue has been corrected.

PCLP-1826 **False positive 9107 for member instantiations**
Message 9107 (header cannot be included in more than one translation unit) was previously being incorrectly issued for member instantiations of out of class definitions. This has been corrected.

PCLP-1841 **Message 923 not issued for `reinterpret_cast`**
Message 923 (explicit cast) was not previously being issued when a `reinterpret_cast` was used. This behavior has been corrected.

PCLP-1894 **False positive 641 for parenthesized enumeration constant in C mode**
Message 641 (implicit conversion of enum to integral type) was being incorrectly issued for parenthesized enumeration constants when appearing in C modules where there was no implicit conversion. For example:

```
//lint -w1 +e641
enum color { COQUELICOT, SMARAGDINE, WENGE };
void foo(enum color);

void bar(void) {
    foo(COQUELICOT);
    foo((COQUELICOT));
}
```

would elicit the incorrect:

```
warning 641: implicit conversion of enum 'color' to integral
    type 'int'
    foo((COQUELICOT));
         ^
```

This issue has been corrected.

PCLP-1908 **False positive 838 for multiple early returns**
A false positive 838 (previous value assigned not used) was sometimes issued in situations where the variable in question was assigned in previous conditional scopes but always returned after assignment. This issue has been corrected.

PCLP-1919 **Improved recognition of volatile assignment as an impurity**
In some contexts, the assignment of a volatile was not treated as an impurity (e.g. for message 522). This issue has been corrected.

PCLP-1909 **False positive 705 for `%jd` conversion specifier with `intmax_t` defined as long long**
When `intmax_t` is defined in terms of `long` instead of `long long` and the sizes of these types are the same, an unexpected 705 (format specifier is nominally inconsistent with argument) is emitted when using the `%jd` conversion specifier with an argument of type `intmax_t`. For example:

```
//lint -w1 +e705
typedef long int  intmax_t;
int printf(const char *, ...);

void foo(int x) {
    printf("%jd", (intmax_t)x);
}
```

elicits the following when the sizes of `long` and `long long` are the same (e.g. -sl8 -sll8):

```
info 705: format '%jd' specifies type 'intmax_t' (aka 'long long')
    which is nominally inconsistent with argument no. 2 of type
    'intmax_t' (aka 'long')
    printf("%jd", (intmax_t)x);
            ~~~   ^~~~~~~~~~~
```

This is because PC-lint Plus treats `long long` as the longest integer type which is a distinct type from `long` even when the sizes are the same. 705 has been softened and will no longer be issued for the `%jd` conversion specifier when the argument type is the same size as `long long`.

PCLP-1910 **Statements without side effects not diagnosed in specific circumstances**
Previously, a statement that lacked a side effect and would be diagnosed with message 522 (highest operation lacks side effect) or 523 (expression statement lacks side effect), was not appropriately diagnosed when the statement in question was either:

1. The first statement immediately following a case label, or
2. The body of a single-statement ranged-based ‘for‘ loop

For example:

```
1 //lint -w1 +e522
2 void foo(int num) {
3     switch(num) {
4         case 1: num; break;
5         default: num; break;
6     }
7 }
```

Message 522 was being issued on line 5 but not line 4. Similarly, in the example:

```
1 //lint -w1 +e522
2 void foo() {
3     int array[10] = {0};
4     for (int i : array)
5         1;
6 }
```

Message 522 was not being issued on line 5. These issues have been resolved.

PCLP-1914 **Message 826 no longer issued for `dynamic_cast`**
Message 826 (suspicious pointer-to-pointer conversion) was being issued for certain types of dynamic casts. As this message is not appropriate for dynamic casts, they will no longer elicit this message.

PCLP-1920 **Improved custodial semantics for reference parameters**
Previously, custody was not being removed from reference parameters which could result in false positive 429 messages. For example:

```
//lint -w1 +e429
//lint -sem(bar,custodial(1))
void bar(char *const &);

void foo(int x) {
    char *s = new char[10];
    bar(s);
}
```

PC-lint Plus was issuing 429 (custodial pointer likely not freed nor returned) for `s` at the end of `foo` despite the fact that it was passed to `bar` which is declared as being custodial via the `-sem` option above. If the parameter for `bar` was not a reference, the message was not issued. This issue has been corrected.

PCLP-1921 **Custodial semantic not properly applied for operator call expressions**
Semantic options used with overloaded operators were not being applied correctly when the overloaded operators were used in operator call expressions. This has been corrected and custodial semantics on overloaded operators will now be honored.

PCLP-1942 **False positive 568/775 for reference member variables**
In some situations a reference to a member variable of signed integer type was being treated as though it could never be negative. For example:

```
//lint -w1 +e568
struct Nifty {
    bool test() { return (i < 0); }
    int &i;
};
```

would result in the incorrect message:

```
warning 568: nonnegative quantity is never less than zero
    bool test() { return (i < 0); }
                         ~ ^ ~
```

A similar situation could elicit undeserved message 775 (nonnegative quantity cannot be less than zero). This issue has been corrected.

PCLP-1943 **False positive 568 for reference types**
Previously, message 568 (nonnegative quantity is never less than zero) would be incorrectly issued for signed integral class members of reference type when compared with zero. This issue has been corrected.

PCLP-1945 **Message 916 now only reports pointer-to-pointer conversions**
Message 916 (implicit pointer assignment conversion) was previously issued for conversions to or from a pointer regardless of whether to other type was also a pointer. This message will now be issued only when both the original and converted types are pointers.

PCLP-1955 **Improperly suppressed messages from library headers**
Previously, some messages could improperly be suppressed for library headers even when they were explicitly enabled for library files. This issue has been corrected.

PCLP-1960 **Messages 900 and 870 not issued when using 2 passes**
Messages 900 (sucessful termination) and 870 (`-max_threads` advisory) were not being issued when exactly two Value Tracking passes were used (e.g. `-vt_passes=2`). This issue has been corrected.

PCLP-1974 **Support for bidirectional pre-loop inference test direction**
In some cases, the test `if (nullptr == p)` was not handled the same as the equivalent `if (p == nullptr)` result in false positive "could be null" diagnostics.

PCLP-1993 **False positive 1924 for substituted non-type template parameters of enumeration type**
Message 1924 (use of c-style cast) was incorrectly emitted when a substituted non-type template parameters of enumeration type was converted. For example:

```
//lint -w1 +e1924
typedef enum { SARCOLINE, MIKADO, SINOPER } Color;
template <typename T> struct S1 { Color eType; };

template <typename T, Color c> struct S2 {
    S1<T> sHeader = {c};
};

S2<int, Color::SINOPER> g_color;
```

would elicit:

```
note 1924: use of c-style cast
    S1<T> sHeader = {c};
                     ^
```

due to an inappropriate C-style cast inserted into the AST to perform the conversion. This issue could also result in other undeserved explicit cast messages and has been corrected.

PCLP-1995 **Reset position indicator character when using `-h2`**
When using the option `-h2`, PC-lint Plus would set the message height to 2 and embed the previously defined position indicator within the source line instead of removing it entirely as PC-lint 9 does in this case. Using `-h2` without specifying a position indicator character will now cause the position indicator to not be emitted.

PCLP-2001 **Improved handling of pointer parameters for message 733**
Previously PC-lint Plus considered function bodies to reside in a narrower scope than the function's parameters which could result in false positive 733 (likely assigning address of local to outer scope pointer) messages. For example:

```
//lint -w1 +e733
void foo(int *pi1) {
    int a;
    pi1 = &a;
}
```

would elicit:

```
 info 733: likely assigning address of local 'a' to outer scope
    pointer 'pi1'
    pi1 = &a;
```

Function parameters are now considered to be in the same scope as the function itself.

PCLP-2013 **Message 2702 now issued regardless of suppression state of 528**
Previously, message 2702 (static symbol declared in header not referenced) was not issued if message 528 (static symbol not referenced) was suppressed. Message 2702 is now issued independent of message 528.

PCLP-2014 **False positive 449**
A false positive 449 (memory was likely previously deallocated) that was sometimes issued in cases where the deallocation order was not correctly considered has been corrected.

PCLP-2015 **False positive 9048 for use of enum constant non-type template parameter**
Message 9048 (unsigned integer literal without 'U') was previously issued when an enum constant was used as a non-type template parameter

```
//lint -w1 +e1924
typedef enum { SARCOLINE, MIKADO, SINOPER } Color;
template <typename T> struct S1 { Color eType; };

template <typename T, Color c> struct S2 {
    S1<T> sHeader = {c};
};

S2<int, Color::SINOPER> g_color;
```

would elicit:

```
note 9048: unsigned integer literal without a 'U' suffix
    S1<T> sHeader = {c};
                     ^
supplemental 897: in instantiation of class template
    'S2<int, SINOPER>' triggered here
S2<int, Color::SINOPER> g_color;
                        ^
```

This issue has been corrected.

PCLP-2029 **Improved handling of angle brackets in message 773**
Message 773 (expression-like macro not parenthesized) was not being issued when a greater-than symbol appeared in a macro definition. For example:

```
1 //lint -w1 +e773
2 #define FOO a < 6
3 #define BAR a > 6
```

Message 773 was issued for line 2 but not line 3. This issue has been corrected and 773 will not be issued in both cases.

PCLP-2068 **Extraneous supplemental message**
In some situations it was possible for PC-lint Plus to issue a supplemental message as the first message without a corresponding primary message. This issue has been corrected.

PCLP-2069 **Don't analyze assembly statements**
In some situations, PC-lint Plus would report on what it believed to be incorrect usage of assembler in in-line assembly statements. Inline assembly will no longer be subject to such critique.

PCLP-2073 **Crash when using allocation semantics in user-defined function semantics**
Previously, PC-lint Plus would crash in certain circumstances when applying user-defined allocation semantics. For example:

```
/*lint -sem(dup, @P == malloc(1P)) */
extern char* dup(const char*);

void alloc(const char *name) {
    if (name == 0) { return; }
    dup(name);
}
```

previously caused PC-lint Plus to crash. This issue has been corrected.

PCLP-2077 **Incorrect null inference for equality check against non-null pointer**
In some cases, using `!=` to compare an unknown pointer against a non-null pointer would create a null pointer inference causing PC-lint Plus to assume the unknown pointer is null. This could in turn lead to false positive messages. This issue has been corrected.

PCLP-2078 **Consistent diagnosis of null pointers for array indexing and pointer arithmetic**
In some cases, a diagnostic related to the use of a null pointer was issued when dereferencing the result of pointer arithmetic but not for the equivalent array indexing syntax. For example:

```
1 void *malloc(unsigned);
2 void foo(int i) {
3     int *ptr = malloc(i);
4     *(ptr + 1) = 0;
5     ptr[1] = 0;
6 }
```

would previously issue a "potential use of null pointer" for line 4 but not line 5. This inconsistency has been corrected.

PCLP-2101 **False positive 9176 for implicit conversion of this pointer to base class pointer**
Previously, PC-lint Plus issued a false positive 9176 (pointer type converted to unrelated pointer type) for implicit conversions of the `this` pointer to a base class pointer. For example:

```
1  //lint -w1 +e9176
2  struct A {
3      void f() { }
4      int i;
5  };
6
7  struct B : public A {
8      void g() {
9          f();
10         i = 2;
11     }
12 };
```

would elicit the messages:

```
9 note 9176: pointer type 'struct B *' converted to unrelated
    pointer type 'struct A *'
        f();
        ^
10  note 9176: pointer type 'struct B *' converted to
    unrelated pointer type 'struct A *'
        i = 2;
        ^
```

This issue has been corrected.

PCLP-2102 **False Positive 743 for wide character constants**
Previously, message 743 (negative character constant) was incorrectly issued for wide character constants. This issue has been corrected.

PCLP-2105 **False positive 527 after switch containing conditional return**
Message 527 (statement is unreachable due to unconditional transfer of control) was previously incorrectly issued in some cases where a switch statement contains a conditional return. This issue has been corrected.

PCLP-2106 **Improved Value Tracking of structures in C**
Value Tracking of structure members was not always recognizing initialization of members via structure initialization in C mode. This issue has been corrected.

PCLP-2108 **False positive 1751 for macro defined in header that expands to anonymous namespace in main source file**

Previously, message 1751 (anonymous namespace declared in a header file) was being issued when a macro defined in header expanded to an anonymous namespace within the main source file. For example, given `test.h` containing:

```
#define NS namespace {}
```

and `test.cpp` containing:

```
#include "test.h"
NS
```

PC-lint Plus would incorrectly report:

```
test.cpp  2  info 1751: anonymous namespace declared in a header file
NS
^
test.h  1  supplemental 893: expanded from macro 'NS'
#define NS namespace {}
           ^
```

This issue has been corrected.

PCLP-2125 **Remove message 504 from MISRA author files**

Message 504 (unusual shift operation) was being used to assist in detecting undefined behavior in some of the MISRA author files. This use of this message for such a purpose is an overreach and its use for that purpose has been eliminated.

PCLP-2127 **False negative 1506**

Message 1506 (call to virtual function within a constructor/destructor) wasn't being issued in some cases where the message was appropriate. This issue has been resolved.

PCLP-2128 **Add POD semantics to Standard C library functions**

POD semantics were not included in the built-in semantics for the Standard C library functions resulting in false negative 1415 (pointer to non-POD passed to function) when using non-POD types with functions like `memmove`, `memcpy`, etc. The appropriate semantics have been added.

PCLP-2140 **False positive 1415 for pointers to void**

Previously, pointers to void were reported via message 1415 (pointer to non-POD type passed to function) for functions expecting pointers to non-POD types. Arguments with an unqualified type of **void *** will no longer be reported as non-POD types.

PCLP-2151 **False positive type alias differences for function template specializations**

Messages 9073, 9094, and 9168 were sometimes incorrectly issued for function template specializations. This issue has been corrected.

PCLP-2153 **Internal error for improper user-defined function return allocation semantics**

Previously, a user-defined function semantic attempting to specify the byte size of an allocation returned by the function in terms of an incomplete type would result in an internal error. This issue has been corrected and such semantics will now be diagnosed with a warning.

PCLP-2155 **False positive 9034 involving compound assignment**
In some cases, the presence of multiple compound assignments could result in false positive 9034 (assigning to narrower or different essential type) messages. This issue has been corrected.

PCLP-2156 **Message 9003 no longer issued for local static variables**
Previously, message 9003 (could define global variable within function) was issued for static variables already local to a function. This issue has been corrected.

PCLP-2157 **False positive 909 and 910 for certain casts**
Previously, in some situations, a false positive 909 (implicit boolean conversion) or 910 (implicit conversion of null pointer constant to pointer) would be issued even if the conversion was the result of a cast (such as `static_cast`). This issue has been corrected.

PCLP-2159 **False positive 1727 for member function of class template specializations**
Message 1727 (function not previously declared inline) was previously incorrectly being issued for member functions of class template specializations. This issue has been corrected.

PCLP-2163 **Strong type of enumerator not recognized when typedef uses incomplete type**
When using the `-strong` option, strong types were not created for typedefs that referenced an enumerator that was not complete until after the typedef definition. This issue has been corrected.

PCLP-2169 **False positive 758 for implicit instantiations**
Message 758 (global tag not referenced) was previously issued implicit instantiations. This issue has been corrected.

PCLP-2179 **Internal error related to message 9027 when issued in C++ code**
If message 9027 (a MISRA C 2012 message) was enabled in C++ mode, there was the possibility of a internal error for templated code. This issue has been corrected.

PCLP-2191 **Structure member initialization status not properly merged after being initialized in both branches of an if statement in C mode**
When a structure member is unconditionally initialized in both branches of an `if` statement in C mode, PC-lint Plus was not correctly recognizing that the member was definitely initialized after the `if` statement. This issue has been corrected.

PCLP-2203 **Improved Value Tracking of unknown function parameters of structure type**
Previously, a function parameter of structure type with unknown value could remain unknown after one of its members was assigned. This issue has been corrected.

PCLP-2208 **Message 1415 issued for type-dependent arguments**
Previously, message 1415 (pointer to non-POD passed to function) was incorrectly issued for type-dependent arguments. This issue has been corrected.

17.2.3 MISRA C 2004 Improvements

PCLP-1874 **Improved support for MISRA C 2004 Rule 13.1**
Message 9236 (assignment operator may not be used within a boolean-valued expression) now supports the exemption in Rule 13.1 which allows assigning a boolean value to a variable.

PCLP-1882 **Improved support for MISRA C 2004 Rules 11.1 and 11.2**
Message 4342 (operand of type cannot be cast to a pointer type) and 4343 (pointer cannot be cast to type) have been replaced with the more precise messages:

- `176` - cannot cast non-pointer non-integer to function pointer
- `177` - cannot cast non-pointer non-integer to object pointer
- `178` - cannot cast function pointer to non-pointer non-integer
- `179` - cannot cast object pointer to non-pointer non-integer

This change eliminates false positives that the previous messages could introduce.

PCLP-1929 **Improved handling of MISRA C 2004 Rule 10.1**
By default, PC-lint Plus will now suppress messages 9225 (integral expression of type 'Type' cannot be implicitly converted to 'Type' because it is not a wider integer type of the same signedness) and 9226 (integral expression of type 'Type' cannot be implicitly converted to 'Type' because it is 'String') for different underlying types of the same size.

PCLP-2009 **Improved support for MISRA C 2004 Rule 10.1**
Message 9225 (integral expression cannot be implicitly converted to type because it is not a wider integer type of the same signedness) was being incorrectly issued when the expression assigned was a dereferenced pointer value. For example:

```
//lint -w1 +e9225
typedef unsigned short uint16_t;

void f(uint16_t * pi) {
    uint16_t r_u16;
    r_u16 = *pi;
}
```

PC-lint Plus would previously issue message 9225 for the assignment to `r_u16`. This issue has been corrected.

PCLP-2018 **Improved support for MISRA C 2004 Rules 6.1 and 6.2**
PC-lint Plus previously classified wide character types as plain character data which could result in false positive messages 9128 (plain character data mixed with non-plain-character data) and 9209 (plain character data used with prohibited operator) when wide character constants were used. For example:

```
/*lint -w1 +e9128*/
typedef unsigned short wchar_t;
void f(wchar_t w) {
    if (w == L'a') { } // false positive 9128
}
```

This issue has been corrected.

PCLP-2037 **Improved support for MISRA C 2004 Rules 6.1/6.2 and MISRA C++ Rule 5-0-11**
Message 9128 (plain character data mixed with non-plain-character data) is now issued for assignments.

17.2.4 MISRA C 2012 Improvements

PCLP-1037 **Improved support for MISRA C 2012 Rule 20.8**

Message 9037 (conditional of `#if` does not evaluate to 0 or 1) is now also issued when the conditional of `#elif` does not evaluate to 0 or 1. For example:

```
1 #if 10
2 #elif 20
3 #endif
```

Message 9037 was previously issued only for line 1 but is not issued for both lines 1 and 2.

PCLP-1528 **Update messages used for Rule 11.7 in au-misra3.lnt**

Message 68 was referenced in au-misra3.lnt to support MISRA C 2012 Rule 11.7. The actual messages corresponding to this rule are 177 and 179. The lnt file has been updated accordingly.

PCLP-1776 **Improved support for MISRA C 2012 Rule 10.3**

Message 9034 (cannot assign essential type to narrower/different essential type) previously did not correctly classify the essential type of compound assignment leading to false positives. For example:

```
//lint -w1 +e9034
void foo(char c) {
    c = c + 1;
    c += 1;
}
```

would elicit the incorrect:

```
note 9034: cannot assign 'signed8' to different essential type
    'character'
    c += 1;
         ^
```

This issue has been corrected.

PCLP-1778 **Improved support for MISRA C 2012 Rule 11.4**

Message 9078 (conversion between object pointer and integer type) was not being issued for boolean conversions. For example:

```
1 //lint -w1 +e9078
2 void foo(_Bool b, int *pi) {
3     b = (_Bool) pi;
4 }
```

Message 9078 is expected on line 3 but was not being issued there. The issue has been corrected. The text of the message has also been adjusted to reflect that the message is limited to *object* pointers.

PCLP-1779 **Improved support for MISRA C 2012 Rule 11.4**
Message 9078 (conversion between object pointer and integer type) was not being issued for implicit conversions. For example:

```
1 //lint -w1 +e9078
2 void foo(void) {
3     void **vp2 = 0x01;
4 }
```

Message 9078 is expected on line 3 but was not being issued there. This issue has been corrected.

PCLP-1780 **Improved support for MISRA C 2012 Rule 11.4**
Message 9078 (conversion between object pointer type and integer type) was previously being issued for conversions involving void pointers which should not be considered object pointers for the purpose of this message. Message 9078 will no longer be issued for conversions involving void pointers.

PCLP-1781 **Improved support for MISRA C 2012 Rule 11.5**
Message 9079 (cast from pointer to void to pointer to ...) was not being emitted for implicit conversions. For example:

```
1 //lint -w1 +e9079
2 void foo(void *pv) {
3     int *pi = (int *)pv;
4     int *pi2 = pv;
5 }
```

Message 9079 is expected on lines 3 and 4 but was only being issued on line 3. This issue has been corrected.

PCLP-1879 **Improved support for MISRA C 2012 Rule 10.1**
While MISRA C 2012 Rule 10.1 prohibits *essentially signed* types from the RHS of a shift operator, an exception is made for non-negative integer constant expressions. In the supporting message 9027, PC-lint Plus was previously only exempting non-negative *integer literals*. For example:

```
//lint -w1 +e9027
void f() {
    1U << (1);
    1U << (1 + 1);
}
```

would elicit:

```
note 9027: a signed value is not an appropriate right
    operand to <<
    1U << (1 + 1);
       ^  ~~~~~~~
```

Non-negative *integer constant expressions* are now exempted from message 9027.

PCLP-1880 **Improved support for MISRA C 2012 Rule 10.1**

Message 9027 (essential type value is not appropriate for operand) was not being issued for the array subscript operator. For example:

```
1 //lint -w1 +e9027
2 void f() {
3     int a[10];
4     a[1 == 1];  //expect 9027
5     a['a'];     //expect 9027
6     a[0];       //expect NONE
7 }
```

Message 9027 was not being issued on lines 4 or 5. This issue has been corrected.

PCLP-1881 **Improved support for MISRA C 2012 Rule 10.2**

Message 9028 (a character value is not an appropriate operand) was not being issued when the RHS of an addition operator had essentially character type but the LHS did not. For example:

```
1 //lint -w1 +e9028
2 enum E { e };
3 void f(char c) {
4     c + e; // expect 9028
5     e + c; // expect 9028
6 }
```

Message 9028 was being issued on line 4 but not line 5. This issue has been corrected.

PCLP-1898 **Improved support for MISRA C 2012 Rule 10.7**

While MISRA C 2012 Rule 10.7 only applies to operators where the usual arithmetic conversions are applied, supporting message 9032 was being issued for shift operators. While shift operators do perform integer promotion, they are not subject to the usual arithmetic conversions and should not be reported by this message.

PCLP-2017 **Do not classify wide character constants as having essentially character type in MISRA C 2012**

MISRA C 2012 does not describe how `wchar_t` should be handled. A strict application of the rules leads to absurd behavior because **L'a'** would be *essentially character* while a variable of type `wchar_t` would be *essentially integral*. For example:

```
/*lint -w1 +e9034 */
typedef unsigned short wchar_t;
wchar_t w = L'w';
```

Would result in the diagnostic:

```
note 9034: cannot assign 'character' to different essential type
    'signed16'
wchar_t w = L'w';
            ^~~~
```

PC-lint no longer classifies non-wide character constants as *essentially character*.

PCLP-2030 **Improved support for MISRA C 2012 Rule 18.8**
Message 9035 (variable length array declared) was not being issued for function parameters of VLA type. For example:

```
//lint -w1 +e9035
void set(unsigned sz, double ary[sz], double val) {
    ary[0] = val;
}
```

Message 9035 was not being issued for the declaration of `ary`. This issue has been corrected.

PCLP-2198 **Incorrect MISRA essential type calculation for modulo operations**
Previously, the incorrect MISRA essential type was sometimes calculated for modulo operations which could result in false positive MISRA violation messages. This issue has been corrected.

PCLP-2220 **Improved support for MISRA C 2012 Rule 11.5**
Added exemption for null pointer constants to message 9079 (conversion from pointer to void to pointer to object type).

17.2.5 MISRA C++ Improvements

PCLP-1952	**Improved support for MISRA C++ Rule 5-3-4** Message 9006 ('sizeof' used on expression with side effect) was sometimes being incorrectly issued in dependent contexts. This issue has been corrected.
PCLP-1989	**Improved support for MISRA C++ Rule 3-1-1** Message 9107 (header cannot be included in more than one translation unit because of the definition of symbol) was previously issued for the out of line definition of class template member function. This is not an ODR violation and the diagnostic will no longer be issued in this case.
PCLP-2031	**Corrections to au-misra-cpp.lnt for MISRA C++ Rules 17-0-1 and 17-0-2** Message 9071 (defined macro is reserved to the compiler) was incorrectly configured to reference Rule 17-0-2 instead of 17-0-1. This issue has been corrected.
PCLP-2117	**Improved support for MISRA C++ Rule 7-3-6** MISRA C++ Rule 7-3-6 allows `using` declarations in function/class scope but this exception wasn't being honored by message 9145 (`using` directive/declaration in header). Message 9145 will no longer be issued for `using` declarations that appear at function or class scope inside of a header.
PCLP-2150	**False positive 9113 for compound assignment** Message 9113 (dependence placed on C++ operator precedence), used to support MISRA C++ Rule 5-0-2, is not intended to be issued for the RHS of an assignment operator unless it itself contains an assignment. This exception was honored for normal assignment but not compound assignment. This has been corrected.
PCLP-2170	**Increased scope of "related types" for message 9176** Message 9176 (pointer type converted to unrelated pointer type) will no longer be issued for any conversions between pointers to related types (types related to each other via inheritance). Previously, downcasts and conversions using e.g. `reinterpret_cast` were diagnosed by this message. MISRA has clarified that this is not the intention of MISRA C++ Rule 5-2-7.

17.2.6 General Improvements

PCLP-1768 **Improved handling when all paths return in `if` statement**

Value Tracking now better recognizes when all paths in an `if` statement will return and uses this information to prevent issuing false positive messages. For example:

```
1  void foo(int *p, bool a) {
2      if (!p) {
3          if (a)
4              return;
5          else
6              return;
7          *p = 2;
8      }
9      *p = 1;
10 }
```

previously resulted in messages 413 and 613 warning about the use of possible null pointers on lines 7 and 9. PC-lint Plus now recognizes that the function will always return when `p` is null and no longer issues the messages.

PCLP-1790 **Improved handling of placement `new` for read-write analysis**

For example:

```
//lint -w1 +e838 +e429
void *malloc(unsigned);
void* operator new (unsigned count, void* ptr);

struct Foo { };

Foo *Allocate() {
    Foo *pool = (Foo *) malloc(sizeof(Foo) * 5);
    if (!pool) { return 0; }

    Foo *item_p = &pool[0];
    item_p = new (item_p) Foo;
    return item_p;
}
```

Would elicit:

```
info 838: previous value assigned to 'item_p' not used
    item_p = new (item_p) Foo;
           ^
supplemental 891: previous assignment is here
    Foo *item_p = &pool[0];
         ^
```

Not considering the semantics of the placement `new`. Message 838 will no longer be issued in such cases.

PCLP-1795 **Improved handling of mutable members in side effects determination**
Assigning to a mutable member in a member function was not previously considered a side effect which could lead to unexpected results. For example:

```
//lint -w1 +e523
struct X {
    void init() { ii = 1; }
    mutable int ii = 0;
};

void foo() {
    X x;
    x.init();
}
```

would elicit the undeserved:

```
warning 523: expression statement involving function 'X::init'
    lacks side effects
    x.init();
    ^
```

Mutable member modifications are now considered to be side effects.

PCLP-1799 **Improved handling of user-defined return value semantics**
Previously, when comparing a pointer return value with literal zero in a user-defined function return semantic, PC-lint Plus would interpret this as implying information about whether the returned pointer could be null. For example:

```
//lint -sem(foo, @p < 10)
int *foo(void);
void bar() {
    int *p = foo();
    *p = 10;
}
```

would cause PC-lint Plus to issue:

```
5  warning 613: potential use of null pointer
    *p = 10;
    ^~
4  supplemental 831: initialization yields
    [-9223372036854775808:36]@0/4? ($6)
    int *p = foo();
    ~~~~~^~~~~~~~~
```

The implication of `@p < 10` being that the return value could be 0 which was interpreted as the possibility of the pointer being null. This behavior has been corrected and such a comparison will not cause nullness information to be implied. Additionally, the lower bound of the inference created for such a pointer return semantic is now non-negative (as opposed to the range given in the above output). To specify that the returned pointer points to less than 10 elements or is null, the semantic option `-sem(foo, @p < 10 || @p == 0)` can be used.

PCLP-1800 **Improved handling of enumeration constants of strongly typed `enums`**
Previously, enumeration constants were never treated as strong types which would lead to unwanted strong type messages when combining a variable of a strongly typed `enum` with an enumeration constant of the same underlying type. For example:

```
//lint -w1 +e63?
//lint -strong(AJX)
typedef enum C {
    ZAFFRE,
    PERVENCHE,
    INCARNADINE,
} color;

extern void setcolor(color c);

void foo(color c) {
    setcolor(INCARNADINE);
    c = PERVENCHE;
    setcolor(c);
}
```

would elicit the messages:

```
warning 632: strong type mismatch: assigning '<non-strong>' to
    'color' in context 'call'
    setcolor(INCARNADINE);
             ^~~~~~~~~~~
warning 632: strong type mismatch: assigning '<non-strong>' to
    'color' in context 'assignment'
    c = PERVENCHE;
      ^ ~~~~~~~~~
```

since the enumeration constants `INCARNADINE` and `PERVENCHE` did not have a strong type classification. The problem with trying to classify enumeration constants as strong types is that multiple typedefs can alias the same enumeration. For example:

```
enum C {
    ZAFFRE,
    PERVENCHE,
    INCARNADINE,
};

typedef enum C color;
typedef enum C shade;
```

Should `ZAFFRE` be strongly classified as `color` or `shade`? PC-lint Plus will now consider enumeration constants to be strong types of the most recent typedef that was seen for the corresponding enumeration (`shade` in the above example).

PCLP-1813 **The `+efreeze` option now always prevents single-line suppressions**
The `+efreeze` and `++efreeze` options will now prevent all future single-line suppressions (`!e#`) from having any effect. Previously, these options did not effect single-line suppressions for messages issued during preprocessing or semantic analysis.

PCLP-1899 **Don't issue 1564 when integer literal is converted/cast**
In the following example:

```
1 //lint -w1 +e1564
2 typedef unsigned char BOOL;
3 #define TRUE ((BOOL) 1)
4 void foo() {
5     if (TRUE) { }
6 }
```

Previously, PC-lint Plus would issue 1564 for line 5. The message has been softened to permit the cast or conversion of the integer literal.

PCLP-1911 **Assume side-effect when passing pointer offset to non-const pointer parameter**
When passing a pointer to a non-const pointer parameter, PC-lint Plus assumes a side-effect but wasn't doing the same when a pointer offset (e.g. `p + 2`) was passed. Pointer offsets are now handled the same as pointers in this case.

PCLP-1932 **Prototype information now included in message 1411**
Message 1411 (member with different signature hides virtual member) now includes the function prototypes in the diagnostic.

PCLP-1934 **"Could be const" messages softened for typedefs from macros**
Messages 818 (parameter of function could be pointer to const), 844 (static storage duration variable could be made pointer to const), and 954 (local variable could be pointer to const) are no longer issued for typedef types defined via a macro. For example:

```
#define MAKE_TYPE(name) struct name##__{int value;};\
    typedef struct name##__ *name
MAKE_TYPE(S1);

int f(S1 s) {
    return s->value;
}
```

would previously result in the message:

```
5  info 818: parameter 's' of function 'f(S1)' could be
    pointer to const
int f(S1 s) {
           ^
```

A similar macro/typedef approach is used by some vendor header files. If a typedef is declared within a macro, it will often be difficult to modify the definition of a variable using that typedef to make it a pointer to const.

PCLP-1951 **Assume modification of initialized arguments when using -fai**
Previously, PC-lint Plus would not assume that initialized variables were modified when passed by non-`const` pointer or reference in a function call. This behavior can result lead to false positive messages. PC-lint Plus will now assume that initialized variables are modified when using `-fai` but will continue to assume that uninitialized variables remain uninitialized.

PCLP-1956 **Clarified message text for message 956**
The text for message 956 (global variable is neither const nor atomic) was changed to replace "global" with one of "static local", "static", "extern", or "global", as appropriate. This is to reduce confusion that can result from referring to e.g. a static local variable as "global".

PCLP-1965 **Add symbol information to message 9103**
Symbol information to was added to the text of message 9103 (identifier with static storage is reused).

PCLP-1967 **Issue message 564 for volatile reads**
In PC-lint 9, this message 564 (variable depends on order of evaluation) was also issued for volatile variables with repeated use within an expression but PC-lint Plus did not carry over this behavior. This message will now be issued in such circumstances.

PCLP-1969 **Improved handling of deallocation tracking in `if` statements**
Previously, a deallocation in a nested `if` statement could incorrectly affect the dealllocation status in the else branch of the enclosing if statement resulting in false positive 2434 (memory was potentially deallocated) messages. This issue has been corrected.

PCLP-1970 **False positive 773 for named casts**
Message 773 (expression-like macro not parenthesized) was being issued for macros whose definition consisted of a named cast. For example:

```
#define TRUE static_cast<int>(!0)
```

would be met with this message when appearing in a C++ module. 773 is no longer issued for named casts.

PCLP-1971 **Exempting logical not from message 1564**
Message 1564 was previously being issued for the logical not operator (`!`) which resulted in undesired diagnostics. This issue has been corrected.

PCLP-1976 **Improved performance for next-statement suppressions**
In some situations, the use of `-e{#}` and `-emacro({#}, ...)` options could result in a significant increase in processing times. This issue has been resolved.

PCLP-1980 **1938 no long issued for `constexpr` variable**
Message 1938 (constructor accesses global data) is no longer issued when the global data referenced is a `constexpr` variable.

PCLP-1985 **Improved handling of structures initialized by functions with using `fai`**
In some cases, structures that were initialized in both branches of an `if` statement would result in inappropriate "possibly uninitialized" diagnostics when the structure's members were accessed outside the `if` statement. This issue has been corrected.

PCLP-1987 **Improved output format for `enum` essential types in essential type messages**

`enum` essential types referenced in MISRA C 2012 essential type messages now include the name of the enumeration, if available, instead of non-deterministic numbering. For example, for the source:

```
/*lint -w1 +e9034 */
typedef enum {GULL, HERON, PLOVER} avian;
unsigned short f(avian x) {
    return x;
}
```

PC-lint Plus previously issued:

```
note 9034: cannot assign 'enum140194779696032' to different
    essential type 'unsigned16'
```

where the actual number could differ between runs. PC-lint Plus will now issue:

```
note 9034: cannot assign 'enum (avian)' to different essential
    type 'unsigned16'
```

PCLP-1992 **False positive 522 in `__asm` statement**

Message 522 (highest operation lack side effects) was sometimes incorrectly issued for `__asm` statements. For example:

```
void foo() {
    __asm("...")
}
```

would elicit this message. Assembly statements are now assumed to have side effects.

PCLP-2002 **Message 785 softened for aggregate initialization with extra braces**

PC-lint Plus previously issued 785 (too few initializers for aggregate) for explicit zero-initialization when extra braces where provided, e.g.:

```
1 //lint -w1 +e785
2 struct X { int a; int b; };
3 void foo() {
4     X x[10] = {0};      // No 785
5     X x[10] = {{0}};    // 785 issued here
6 }
```

would result in message 785 for line 5 but not line 4. The exemption for explicit zero-initialization has been extended to initializers with extra braces.

PCLP-2205 **Improved support for C++17 `constexpr if`**

Message 774 (boolean condition always true/false) is no longer issued for the condition of a `constexpr if`. Additionally, messages 9012 (body should be a compound statement), 548 (if statement has no body or else), and 9138 (null statement not on line by itself) will no longer be issued for a discarded statement.

PCLP-2006 **Recognition of aggregate initialization evaluation order in C++11**
Previously PC-lint Plus would issue 446 (side-effect in initializer list) for all side effects present in an aggregate initialization. The stated purpose of the message is to diagnose cases where the unspecified order of evaluation of the side effects can result in unspecified values for aggregate elements. In C++11, the order of evaluation of aggregate initialization is specified so the message is no appropriate in such cases. Message 446 will no longer be issued in C++11 mode and higher.

PCLP-2024 **Support flexible array members in base classes**
PC-lint Plus now supports the use of flexible array members in base classes. This is illegal in Standard C++ and as such will draw an error message (4293) but will otherwise be accepted. If your compiler supports this feature, you can safely suppress this message.

PCLP-2039 **Clarifications to text of message 9128**
Message 9138 (plain char type mixed with type other than plain char) treats all character constants as character type as per MISRA C 2004 Rules 6.1 and 6.2 and MISRA C++ Rule 5-0-11. Neither the message text nor the description made it clear that this was the case. The message text has been updated to better reflect the issue diagnosed.

PCLP-2050 **Improved `pclp_config` support for macro definition that contain spaces and quotes in Visual Studio configurations**
When imposter output contained the definition of a macro using the `/D` option whose definition contained both spaces characters and quotes and this output was later processed by `pclp_config` to generate a project configuration for a Visual Studio compiler, the resulting transformed option would not always be correct which could lead to errors. `pclp_config` and `compilers.yaml` have been updated to better support such macro definitions.

PCLP-2074 **Improved Value Tracking handling of `while` statement condition variable declarations**
The handling of condition variable declarations in `while` statements has been improved which corrects some false positive Value Tracking messages related to the use of such constructs.

PCLP-2075 **Improve application of dynamic allocation semantics**
Previously, dynamic allocation semantics were not applied to unknown pointers and cases where PC-lint Plus could not infer information about the size of the allocated region. For example:

```
/*lint -sem(my_alloc, @P == malloc(1P)) */
int* my_alloc(void *);

void foo(int *p1) {
    int *p2 = my_alloc(p1);
    p2[1] = 0;
}
```

a warning about custodial pointer `p2` not being freed would be expected here but previously was not issued. This has been corrected.

PCLP-2081 **Message 1793 no longer issued for functions with an rvalue reference qualifier**
Previously, message 1793 (invoking non-const member function on a temporary) was issued for functions with an rvalue reference qualifier. While technically correct, a message in such cases isn't useful since the concern articulated by the description of the message (that any changes made by the function may be inadvertently lost) doesn't apply here since the function is explicitly declared as working on rvalues. Message 1793 will no longer be issued in such cases.

PCLP-2088 **Improved handling of premature termination due to stack overflow**
Previously, when PC-lint Plus experienced a stack overflow, it would abort without a a crash message. PC-lint Plus will now provide a crash message with troubleshooting information in the event of a stack overflow.

PCLP-2098 **Improved Value Tracking for unconditional assignment in both branches of an `if` statement**
In the following example:

```
1 int f(int arg) {
2     unsigned x = 0;
3     if (arg > 0)
4         x = 20;
5     else
6         x = 55;
7
8     return 10u % x;
9 }
```

PC-lint Plus would previously issue 414 (possible division by zero) on line 8, incorrectly assuming that `x` could still be zero at this point. PC-lint Plus will now recognize that this is not the case.

PCLP-2152 **Improved handling of friends and templates for 9004**
Message 9004 (object/function previously declared) will no longer be issued for friend declarations, templates, or template specializations.

PCLP-2161 **Improved handling of misspelled parameterized suppression options**
PC-lint Plus will now diagnose invalid suppression options that were not previously reported such as `-emarco(123, A)` instead of `-emacro(123, A)`.

PCLP-2164 **Add symbol parameter to null pointer dereference messages when available**
Messages 413 (likely use of null pointer) and 613 (potential use of null pointer) now include the corresponding symbol when one is available.

PCLP-2166 **Message 904 issued multiple times for repeated declarations**
Message 904 (return before end of function) was previously issued once for each declaration of the offending function. This message will now be issued only once regardless of how many declarations are present.

PCLP-2178 **Suppress message 1788 for variables marked as unused**
Message 1788 (variable is referenced only by its constructor/destructor) will no longer be issued for variables that have been marked as unused (such as by the C++17 `[[maybe_unused]]` attribute).

PCLP-2185 **Improved error handling for invalid size and alignment options**
The 686 (suspicious option) message given for incorrect use of the `-s` and `-a` options now more clearly communicates when an option is being processed as a size or alignment option.

PCLP-2195 **Improved handling of possibly null information in user-defined return semantics**
The following improvements were made related user-defined function return semantics (`-sem`):

- The use of a `malloc` semantic to indicate a function returns a malloc'd pointer no longer implies a potentially null return value. This was a divergence from PC-lint 9 behavior and has been corrected.
- The documentation for the `fnr` flag states that functions with a custom return semantic are not effected by the flag. This provision was not being honored and has been corrected.
- Inferencing of the size of a buffer is now distinct from inferencing nullness in a return semantic. Using `==` to compare the pointer return value to `0` will imply the return value may be null while all other operations are used to inference the size of the pointed to buffer.

PCLP-2200 **Improved handling of buffers used with placement `new`**
When an allocated buffer is used in a placement `new` operation, the custodial status and allocation method is now removed from the buffer which eliminates false positive complaints about unallocated custodial pointers and inappropriate allocation methods being used.

PCLP-2214 **Added `env-html.lnt`, `env-html.js`, and `env-xml.lnt` files**
Added updated configuration files for HTML and XML message output.

17.2.7 New Features

PCLP-1531	**Support for IAR compilers and IAR Workbench** `pclp_config` now supports configuration for IAR Embedded compilers, see 2.3.5 Creating a compiler configuration for IAR Embedded compilers for details. PC-lint Plus also supports integration with IAR Embedded Workbench via the `env-iar.lnt` file, see 2.3.8 Integrating PC-lint Plus with IAR Embedded Workbench for instructions on how to perform the integration.
PCLP-1889	**New message 3450 - subtracting member from 'this' pointer** The new message 3450 diagnoses the use of `-` when `->` was intended in expressions involving the `this` pointer and an implicit access of a member of the same class, e.g. `this-value` instead of `this->value`. See the description of this message for additional information.
PCLP-2051	**New `fbe` flag option and backslash escapes** PC-lint Plus now allows special characters to be escaped in options by preceding them with a backslash when the `fbe` flag is ON (it is OFF by default): `{ } ( ) [ ] ! , " \` The `fbe` flag can be turned ON and OFF between options.
PCLP-2054	**Added support for MISRA C 2012 AMD-1 Rule 12.5** We now support MISRA C 2012 AMD-1 Rule 12.5 using the existing message 682 (sizeof applied to parameter of function whose type is a sized array) and the new message 882 (sizeof applied to parameter of function declared an incomplete array type). See the new au-misra3-amd1.lnt file for details.
PCLP-2055	**Added support for MISRA C 2012 AMD-1 Rule 21.13** We now support MISRA C 2012 AMD-1 Rule 21.13, see the new au-misra3-amd1.lnt file for details.
PCLP-2057	**Added support MISRA C 2012 AMD-1 Rule 21.15** We now support MISRA C 2012 AMD-1 Rule 21.15 with the new message 857 (incompatible pointer arguments to memcpy/memmove/memcmp). See the new au-misra3-amd1.lnt file for details.
PCLP-2058	**Added new message (9098) to support MISRA C 2012 AMD-1 Rule 21.16** The new message 9098 (pointer argument to function does not point to a pointer type or an essentially signed, unsigned, boolean, or enum type) provides support for MISRA C 2012 AMD-1 Rule 21.16. See au-misra3-amd1.lnt for more information.
PCLP-2059	**Add partial support for MISRA C 2012 AMD-1 Rule 21.17** The new au-misra3-amd1.lnt file provides partial support for the (undecidable) MISRA C 2012 AMD-1 Rule 21.17.
PCLP-2083	**Allow the C++17 fallthrough attribute to suppress messages 616 and 825** The C++17 `[[fallthrough]]` attribute can now be used to suppress messages 616 (control flow falls through to next case without an intervening comment) and 825 (control flow falls through to next case without an intervening -fallthrough comment).
PCLP-2132	**The `-help` option now responds to message numbers** A message number can now be provided as an argument to the `-help` option to obtain information about the message and the message description.

PCLP-2201 **New exit command added to Value Tracking debugger**
A new `exit` command has been added to the Value Tracking debugger which will cause PC-lint Plus to terminate.

17.2.8 Documentation Enhancements

PCLP-1926 **Add "Flow of Execution" section (1.2) to Reference Manual**
This new section more clearly articulates what the module and global wrap-up phases are, when they occur, and the relationship between the options `-max_threads`, `-unit_check`, `-vt_passes` and the different phases of execution.

PCLP-1949 **Fix runaway text in Reference Manaual**
The text in section 14.11 "Torture Testing Your Code" in the Reference Manual ran outside the right margin of the page. This issue has been corrected.

PCLP-1966 **Update description of message 9049**
For the purpose of message 9049 (increment/decrement operation combined with other operation with side-effects), a function call is always considered to have side effects, as per the corresponding MISRA Rule. While this behavior was correctly implemented, this behavior was not explicit in the message description.

PCLP-2008 **Reference the `flf` flag in the Value Tracking section of the Reference Manual**
Previously, the description for the `flf` flag referenced the Value Tracking section but the Value Tracking section did not contain a reference back to the `flf` flag. The `flf` flag is now referenced in the Value Tracking section.

PCLP-2012 **Improvements to descriptions of select error messages**
The descriptions of errors 18, 21, 64, 131, and 322 were updated to correct outdated or incorrect verbiage.

PCLP-2016 **`fwc` flag removed from manual**
References to the obsolete `fwc` flag have been removed from the Reference Manual.

PCLP-2049 **Miscellaneous documentation corrections**
In Section 4.5.1 of the Reference Manual, the option `-sp` was not included in the list of size options but the unsupported `-smpD` option was. In section 4.3.3 (Message Presentation), the default format for the `-format_summary` contained an extraneous character '`t`'. Both issues have been corrected.

PCLP-2172 **Clarify that `-egrep` doesn't match text injected via `+typename`**
The description of the `-egrep` option has been updated to clarify that the text that it matches does not include text that was injected by using the `+typename` option.

PCLP-2210 **Update description of message 916 to remove MISRA C++ support statement**
MISRA C++ Rule 5-2-7 is supported by message 9176 but a support statement for this rule is still present in the description of message 916 which was used to support this rule in an older version. This support statement has been removed.

18 Open Source Declarations

PC-lint Plus incorporates several pieces of Open Source Software. The following declarations are made in compliance with the respective licenses.

- `LLVM/clang`
  - Attribution: C and C++ front-end support is provided by the LLVM and clang projects.
  - License: `University of Illinois/NCSA Open Source License`
- `PCRE`
  - Attribution: Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England.
  - License: PCRE2 is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

    Release 10 of PCRE2 is distributed under the terms of the "BSD" license, as specified below. The documentation for PCRE2, supplied in the "doc" directory, is distributed under the same terms as the software itself. The data in the testdata directory is not copyrighted and is in the public domain.

    The basic library functions are written in C and are freestanding. Also included in the distribution is a just-in-time compiler that can be used to optimize pattern matching. This is an optional feature that can be omitted when the library is built.

    THE BASIC LIBRARY FUNCTIONS
    Written by: Philip Hazel Email local part: ph10 Email domain: cam.ac.uk
    University of Cambridge Computing Service, Cambridge, England.

    PCRE2 JUST-IN-TIME COMPILATION SUPPORT
    Written by: Zoltan Herczeg Email local part: hzmester Emain domain: freemail.hu

    STACK-LESS JUST-IN-TIME COMPILER
    Written by: Zoltan Herczeg Email local part: hzmester Emain domain: freemail.hu

    THE "BSD" LICENSE
    Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

    1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
    2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
    3. Neither the name of the University of Cambridge nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

19 Bibliography

[1] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[2] Dan Saks. *const T vs. T const*. www.dansaks.com,Published Articles, 1999.

[3] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates – The Complete Guide*. Addison-Wesley.

[4] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.

[5] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[6] David A. Spuler. *C++ and C Debugging, Testing and Reliability*. Prentice Hall, 1994.

[7] Scott Meyers. *Effective C++ Third Edition*. Addison-Wesley.

[8] Herb Sutter. *Exceptional C++*. Addison-Wesley.

[9] Allen I. Holub. *Enough Rope to Shoot Yourself in the Foot*. McGraw Hill, 1995.

[10] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, First Printing 1990, Reprint w/corrections 1992.

[11] ISO/IEC. *14882:1998 - Programming Languages - C++*. American National Standards Institutue, 1998.

[12] ISO/IEC. *14882:2003 Programming Languages C++*. International Standard, 2003.

[13] ISO/IEC. *14882 C++ Standard Core Language Defect Reports*.

[14] Tom Cargill. *C++ Gotchas*. Presented at C++ World, November 1992.

[15] Andrew Koenig. *Check list for Class Authors*. 1992 Nov 1.

[16] Tom Cargill. *C++ Programming Style*. Addison-Wesley, 1992.

[17] The Motor Industry Research Association. *Guidelines of the Use of the C Language in Vehicle Based Software (MISRA)*. Warwichshire, 1998.

[18] Dan Saks. *C++ Gotchas!* Saks and Associates.

[19] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley.

[20] Scott Meyers. *Effective Modern C++*. O'Reilly, 2015.

[21] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 1978.

[22] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Pearson, 5th edition, 2002.

[23] ISO/IEC. *9899:1999 Programming languages - C*. American National Standards Institute, 1999.

[24] INCITS/ISO/IEC. *14882-2011 Programming languages - C++*. American National Standards Institute, 2012.

[25] Robert Ward. *Debugging C*. Que Corporation, 1986.

[26] Rex Jaeschke. *Portability and the C Language*. Hayden Books, 1989.

[27] Les Hatton. *Safer C*. McGraw-Hill, 1995.

[28] Peter Van Der Linden. *Expert C Programming - Deep C Secrets*. Prentice Hall, 1994.

[29] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[30] Bruce Eckel. *C++ Inside and Out*. Osborne / McGraw-Hill, 1992.

[31] S. Hekmatpour. *C++: A Guide for Programmers.* Prentice Hall, 1992.

[32] T. Plum and D. Saks. *C++ Programming Guide.* Plum Hall, 1991.

[33] Bjarne Stroustrup. *The C++ Programming Language., 2nd Ed.* Addison-Wesley, 1992.

[34] Larry Reznick. *Tools for Code Management.* R&D Books, 1996.

[35] Andrew Koenig and Barbara Moo. *Ruminations on C++.* Addison-Wesley.

[36] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards (101 Rules, Guideline, and Best Practices).* Addison-Wesley.

[37] Bil Lewis and Daniel J. Berg. *Multithread Programming with Pthreads.* Sun Microsystems Press.

[38] The Motor Industry Research Association. *MISRA-C:2004 Guidelines for the use of the C Language in critical systems.* 2004.

[39] The Motor Industry Software Reliability Association. *MISRA-C++:2008 Guideline for the use of the C++ Language in critical systems.* The Motor Industry Research Association, 2008.