U S E R ' S G U I D E

STATIC

Version 1.2

Static Analyzer

SOFTWARE RESEARCH, INC.

This document property of:

Name:_____

Company:_____

Address:_____

_____

Phone_____

SOFTWARE RESEARCH, INC.

625 Third Street
San Francisco, CA 94107-1997
Tel: (415) 957-1441
Toll Free: (800) 942-SOFT
Fax: (415) 957-0730
E-mail: support@soft.com
http:www.soft.com

Documentation: Ann Kuchins

(Last Update November 19, 1997)
documentation/user-manuals/unix/advisor/97advisor.book

Table of Contents

List of Figures

Preface

This preface explains how this user's guide is organized.

Congratulations!

By choosing the TestWorks integrated suite of testing tools, you have taken the first step in bringing your application to the highest possible level of quality.

Software testing and quality assurance, while becoming more important in today's competitive marketplace, can dominate your resources and delay your product release. By automating the testing process, you can assure the quality of your product without needlessly depleting your resources.

Software Research, Inc. believes strongly in automated software testing. It is our goal to bring your product as close to flawlessness as possible. Our leading-edge testing techniques and coverage assurance methods are designed to give you the greatest insight into your source code.

TestWorks is the most complete solution available, with full-featured regression testing, coverage analyzers, and metric tools.

Audience

This manual is intended for software testers who are using *STATIC* tools. You should be familiar with the X Window System and your workstation.

Format of Chapters

This manual is organized to aid you after installation has been completed (See the *Installation Instructions* if you are trying to install.).

This manual is divided into the following sections:

Chapter 1	*INTRODUCTION TO STATIC* explains the basic functions of *STATIC*.
Chapter 2	*QUICK START* is a tutorial and shows step-by-step how to run a basic STATIC test session.
Chapter 3	*STATIC GUI OPERATION* covers the basic X Window System graphical user interface operations of *STATIC*.
Chapter 4	*MESSAGES* details all of the error message *STATIC PRODUCES*.
Chapter 5	*LIBRARIES* explains what library modules are, how they are used to describe libraries, and how to use the alternative library object module.
Chapter 6	*LINT OBJECT MODULES* defines Lint Object Modules, how they are used, and how to produce one.
Chapter 7	*SPECIAL FEATURES* discusses how *STATIC* checks for the following: out-of order expressions, formats, indentations, consts, and volatiles.
Chapter 8	*LANGUAGE EXTENSIONS* describes generally-accepted, non-K&R extensions to the C language which have been optionally incorporated into *STATIC*.
Chapter 9	*PREPROCESSOR* discusses *STATIC ANSI* and non-ANSI as well as include processing.
Chapter 10	*ADDITIONAL NOTES* discusses how the size of scalars may affect your report results.
Chapter 11	*COMMON PROBLEMS AND APPLICATIONS* describes how to handle common problems and how to use *STATIC* in a practical manner.

Identifying Special Text

This section explains the typographical conventions that are used throughout this manual.

boldface Introduces or emphasizes a term that refers to TestWorks' window, its sub-menus and its options.

italics Indicates the names of files, directories, pathnames, variables, and attributes. Italics is also used for manual and book titles.

"Double Quotation Marks"

Indicates chapter titles and sections. Words with special meanings may also be set apart with double quotation marks the first time they are used.

`courier` Indicates system output such as error messages, system hints, file output, and *CAPBAK/X*'s keysave file language.

`Boldface Courier`

Indicates any command or data input that you are directed to type. For example, prompts and invocation commands are in this text. (For instance, **stw** invokes TestWorks.)

CHAPTER 1

Introduction to STATIC

This chapter explains the *STATIC* basics. You will learn the basic functions of *STATIC*, how it can help you, and its role in a Quality Assurance activity.

1.1 Static Analysis

STATIC finds quirks, idiosyncrasies, glitches and bugs in C programs. The purpose of such analysis is to determine potential problems in C programs prior to integration or porting, or to reveal unusual constructs that may be a source of subtle and, as 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.

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

```
half(x)
    double x;
    {
    return x / 2;
    }

main()
    {
    double n;

    n = half(10000);
    printf( "%d", n );
    }
```

The diagram below illustrates the *STATIC* process.

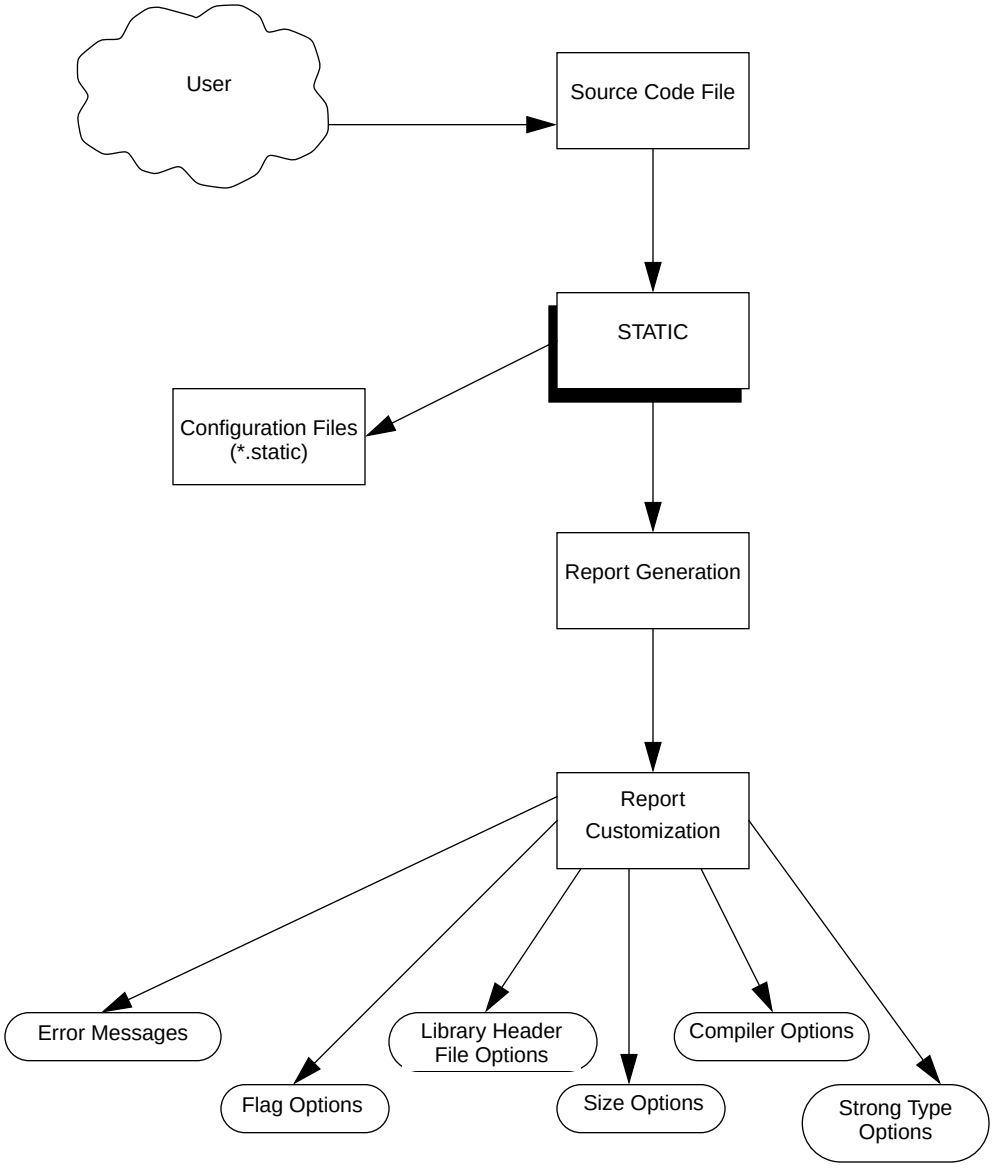

FIGURE 1 STATIC System Flow Chart

As far as many compilers are concerned, it is a valid C program. However, it has a number of subtle errors and question marks that will be reported upon by *STATIC*. The `return` statement of `half()` shows a `double` being returned but half() is typed int (by default). Therefore x/2 is truncated to integer before returning. Is this what the programmer wanted? Or did he/she forget a declaration somewhere? This is reported upon by *STATIC* because the assignment (or implied assignment in this case) loses information. If the programmer really wants to return an integer then a cast should be used as in:

```
(int)(x/2).
```

Another problem is that `half()` is called with an `int` argument whereas it expects a `double`. These kinds of errors get by many compilers but *STATIC* will report on a mismatch (in number or type) of argument lists. *STATIC* has a number of options to lower its sensitivity to a type mismatch (See Section 3.7.1 - "Error Messages Options" on page 32.) and also an option to indicate that some functions have variable arguments (See Section 3.7.2 - "Flag Options" on page 38.).

As a third problem, the format specifier (`%d`) implies an `int` whereas a `double` is provided as argument. `printf` is one of several functions about which *STATIC* has built-in knowledge. For the most part, *STATIC* obtains information about library functions by processing compiler-provided header files.

1.2 Language Definition

STATIC assumes the ANSI definition of C and supports K&R where it does not conflict with ANSI. It also supports common extensions to the standard especially for specific compilers. See Section 19.2 for non-ANSI extensions and Chapter 20 for preprocessor information.

The Kernighan & Ritchie (K&R) description of the C programming language [1] has served as a de facto standard ever since its publication in 1978. An excellent exposition of this standard as well as a thorough description of current practice within the C community is provided by Harbison & Steele [3].

Over the past several years, the ANSI (American National Standards Institute) C committee (X3J11) has developed a C standard [2] that is largely upward compatible with K&R (one of its major tenets was to "not break working code").

At this writing, the work of the ANSI committee has drawn to a close and it seems clear that their efforts are successful. Most major vendors have adopted the standard or have at least indicated intentions of evolving toward the standard. ISO (the International Standards Organization) has so far adopted the ANSI work and authors K&R and H&S have produced subsequent editions of their respective works based on the ANSI standard.

1.3 Main System Features

- Reads any compilable C language source code file.
- Allows you to analyze entire groups of source code files whose names match some sort of pattern.
- Automatically computes a message report.
- Six kinds of messages can be reported:
  - Syntax Errors.
  - *STATIC* Internal Errors.
  - Fatal Errors.
  - Warning Messages.
  - Informational Messages.
  - Elective Note Messages.
- Allows you suppress or activate available error messages as well as:
  - Flags that give directives which effect *STATIC*'s behavior.
  - Library headers that explain how libraries are passed to *STATIC*.
  - Size options.
  - Compiler vendor switches and compiler feature options.
  - `typedef`-based type-checking options.
- Functions accessed through a X Window System graphical user interface (GUI).

CHAPTER 2

Quick Start

This chapter is a tutorial that shows step-by-step how to run a basic *STATIC* session, including invoking *STATIC*, loading a source code file, analyzing a resulting report, and suppressing error messages. If you are an advanced *STATIC* user, you may skip this chapter. This chapter is intended for beginning and intermediate users.

2.1 Instructions

It is recommended that you complete the instructions in this chapter *before* continuing to other sections. This chapter will give you a feel for how to use *STATIC*.

For best results, follow the instructions very carefully. When you have completed this chapter, you should be familiar with the main activities involved in using *STATIC*, including selecting a source code file to analyze, analyze the resulting report, suppressing an error message, and then viewing the impact.

If you are a first-time *STATIC* user, this chapter is best used if you make reference to the appropriate chapter for further operational instructions (See CHAPTER 3 - "STATIC GUI Operation" on page 19.).

If you have the **Xplabak** utility (playback utility for *CAPBAK*^{TM}) you can run the supplied `static.ksv` file to see an example of how this session works. The instructions are at the end of this chapter.

2.1.1 STEP 1: Setting Up STATIC

You should start with the screen organized in a particular way, as shown in the figure (See Figure 2 "Setting Up the Display (Initial Condition)" on page 6.).

Initialize an xterm-type window by using the mouse to click on **New Windows** or issuing the command **`xterm &`** from an existing window. The xterm window will serve as the *STATIC* invocation window.

Move the window to the upper left of the screen. Go to the **$SR/demos** directory. The **demos** directory is supplied with the product, and it consists of a source code file named `xcalc.c`. This tutorial will make use of this file.

When initiating this quick start session, your display should look like this:

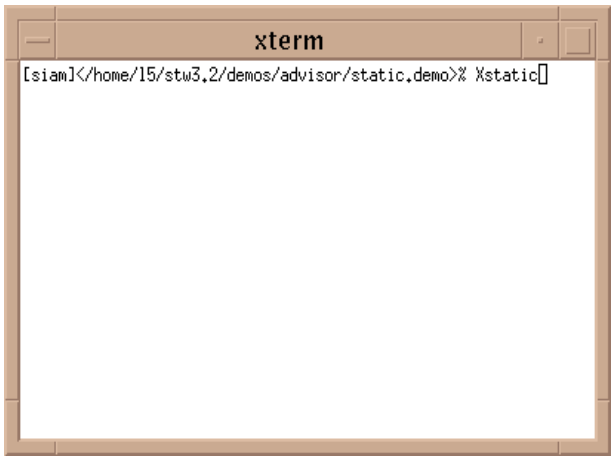

FIGURE 2 Setting Up the Display (Initial Condition)

2.1.2 STEP 2: Invoking STATIC

Now, invoke *STATIC*:

1. Position the mouse so that it is located in the invocation window.
2. Activate it by clicking the mouse button on it. This window becomes the main control window. During your session, all status messages and warnings are displayed in this window.
3. Start *STATIC* from your working directory by typing in:

   ```
   Xstatic
   ```

4. When you type in the command, the **Main STATIC** window pops up. All operations for *STATIC* can be performed from this window.
5. Move the **Main** window to the lower right of the screen. You can move a window by clicking on its title bar and dragging it.
6. If you want to start over, you can terminate from the **Main** window, by clicking on the **File** pull-down menu and selecting **Exit**.

After invoking *STATIC*, your display should look like this:

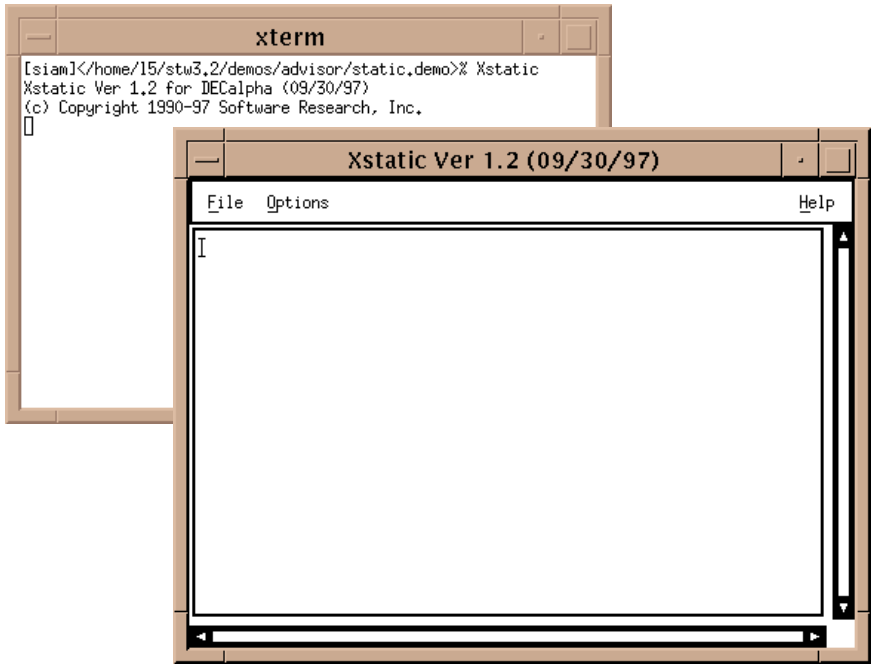

FIGURE 3 Invoking STATIC

2.1.3 STEP 3: Selecting a Source Code File

To obtain a report for a source code file, all you have to do is select any compilable file. *STATIC* is a static code analyzer, so you do not have to do anything special to a program's code. For this demo, select the file named `xcalc.c`:

1. Click on the **File** pull-down menu.
2. Select the **Load Single File** option.
3. A file selection dialog box pops up.
4. To select `xcalc.c`, do one of three things:
   - Double click on **xcalc.c** in the File selection window, or
   - Highlight **xcalc.c** in the File selection window or type in the file name in the Selection entry box and click on **OK**, or
   - Highlight or type in **xcalc.c** and press the **<ENTER>** key.
5. *STATIC* automatically processes the source code and generates a report.

When selecting a source code file, your display should look like this:

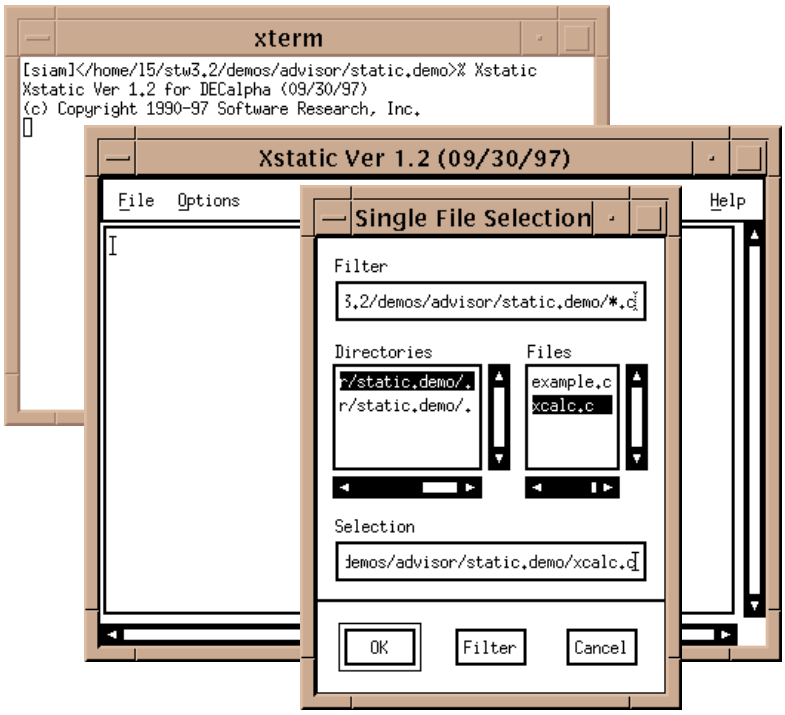

FIGURE 4 Selecting a Source Code File

2.1.4 STEP 4: Analyzing the Report

After selecting the source code file, *STATIC* automatically processes the source code file. The resulting report consists of messages which identify program errors and potential hot-spots found.

You can use the scroll bars to move up and down and side to side.

It contains the following information:

1. Module Name. The path and the source code file name are indicated.
2. Program Statement.
3. Line Number. For each message, the source code file line number that it applies to is indicated.
4. The Type of Message. The type of message for the program statement is indicated. *STATIC* reports the following messages:
   - Error Messages
   - Internal Messages
   - Fatal Messages
   - Warning Messages
   - Informational Messages
   - Elective Notes

   For further information, please refer to the appropriate chapter (See CHAPTER 4 - "Messages" on page 93.).
5. Number and Message. Each message type is identified by a number and brief description.

When analyzing the report, your display should look like the one below:

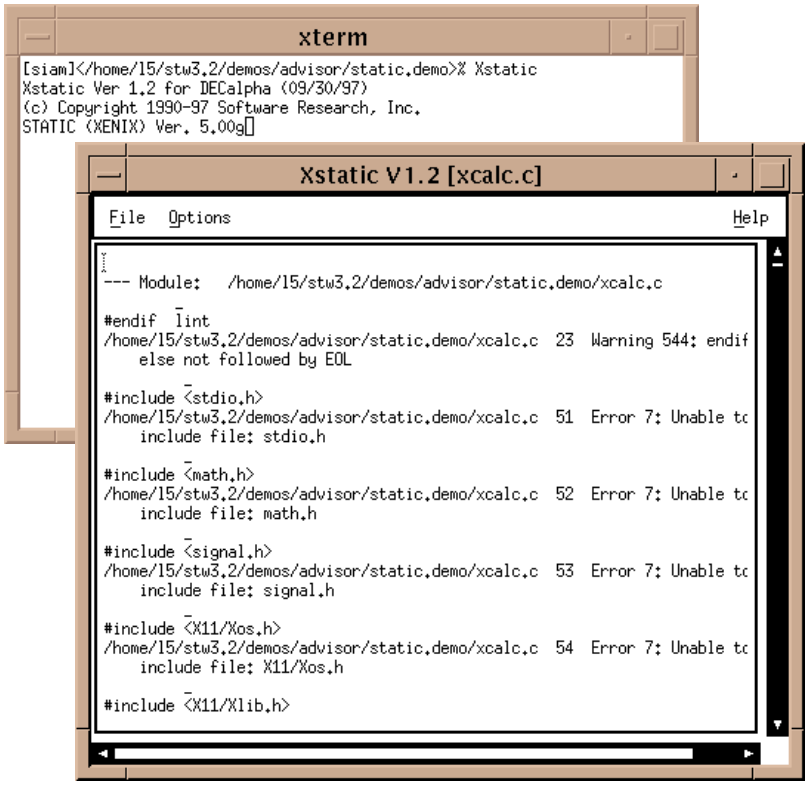

FIGURE 5 Analyzing the Report

2.1.5 STEP 5: Modifying the Report

As you may have noticed, the report is quite substantial. Although many of the messages will identify error-prone code, there will be times when the code identified really isn't error-prone. It could simply be a matter of programming style. For this reason, *STATIC* allows you to suppress error messages.

For the purpose of this example, you are going to suppress the first error messages listed in the report: 544 and 7. If you want to know more about these messages, you can refer to the appropriate chapter for their meaning (See CHAPTER 4 - "Messages" on page 93.).

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu and select **Error**.
3. The **Error Options** window pops up. On the left side of the window, the available options are listed. On the right, the **Error Options Set** window lists the default options.
4. The `-/+e#` option allows you to suppress or activate particular messages.
5. Position the mouse pointer so it is in the specification region and click. A cursor should appear.
6. To suppress message 544, type in:

   ```
   -e544
   ```

   and then click on the **Add** button.
7. `-e544` should appear at the bottom of the **Error Option Set** window.
8. To suppress message 7, do the same as you did for message 544 in step 6.
9. To exit the window, click on the **Close** button.

When suppressing error messages, your display should look like the one below:

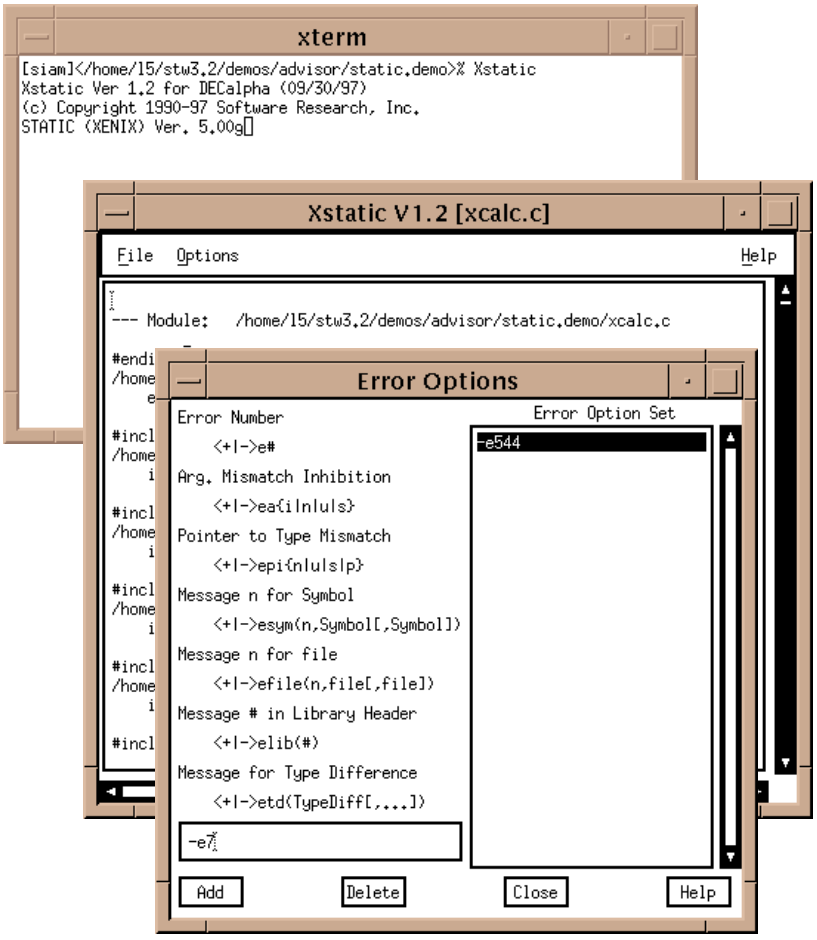

FIGURE 6 Suppressing Error Messages

2.1.6 STEP 6: Activating Modifications

For the report to reflect these changes, you must reload the `xcalc.c` file:

1. Click on the **File** pull-down menu.
2. Select the **Load Single File** option.
3. When the file selection dialog box pops up, select `xcalc.c`.
4. *STATIC* automatically re-processes the source code file to generate another report.
5. This time, however, messages 544 and 7 are gone.

After you make modifications and re-load a source code file, the report should be changed:

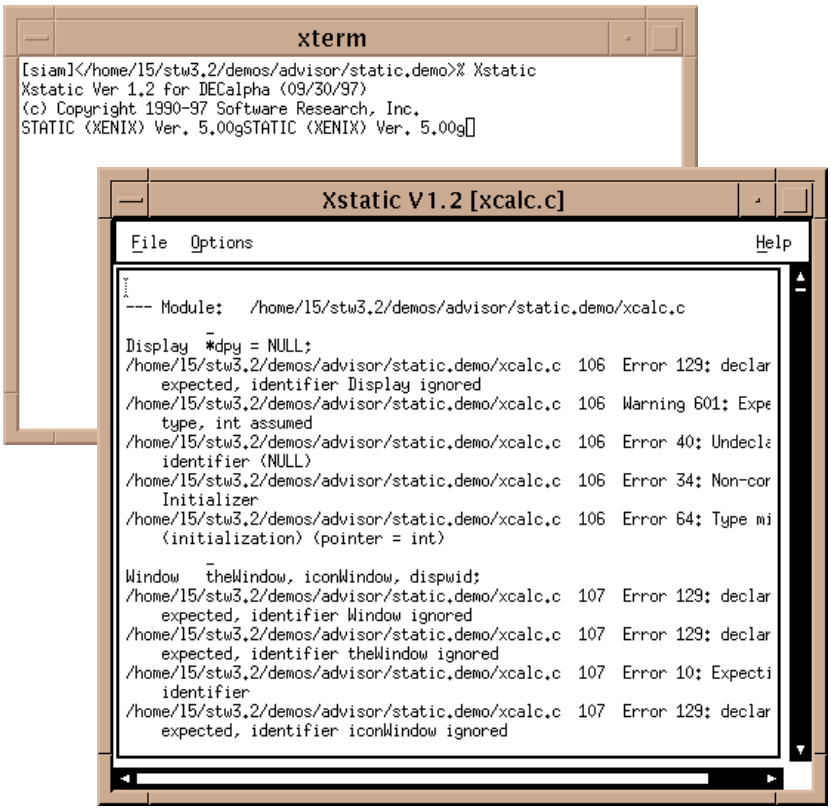

FIGURE 7 Re-loading a Source Code File

2.1.7 STEP 7: Sign Off and Cleanup

To complete this session:

1. Click on **Main** window's **File** pull-down window.
2. Select **Exit**.
3. Because modifications were made to the default error option settings, you will be prompted with a dialog box if you want to save those changes.
4. Since this is a demo, do not save this modifications. Click on **No**.

When exiting this *STATIC* session, your display should look like this:

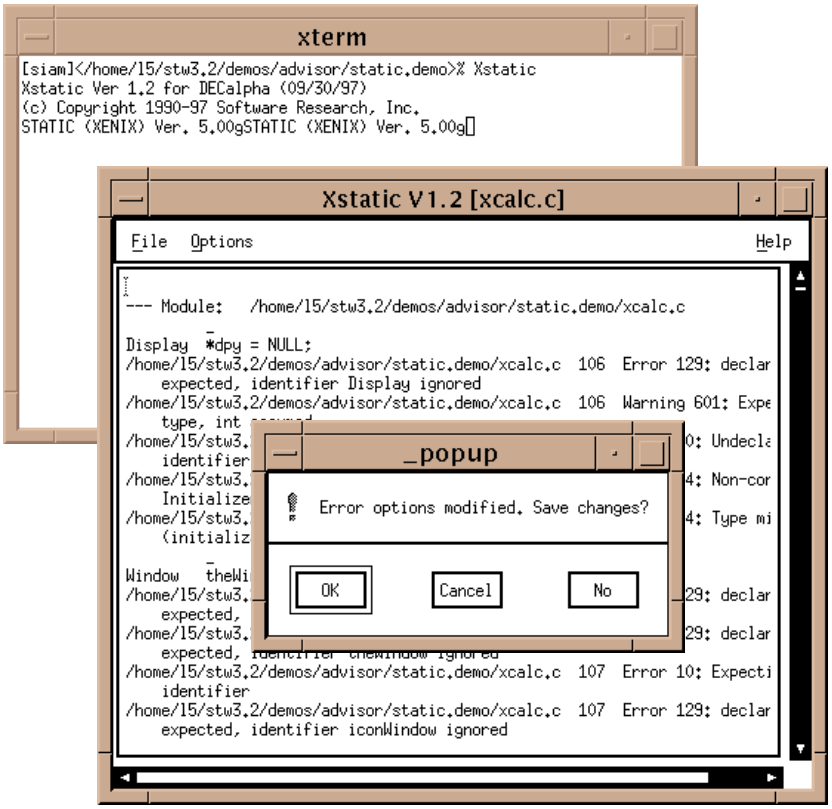

FIGURE 8 Completing a STATIC Session

2.2 Summary

If you successfully completed the preceding 7 steps, you've seen and practiced the basic skills you need to use *STATIC* productively. In this chapter you should have learned how to invoke *STATIC*, how to load single file, how to analyze a report, and how to suppress error messages.

For best learning, you may want to

- Repeat STEPS 1 - 7 with your application.
- Turn to the appropriate chapters to learn more about other kinds of modifications (See CHAPTER 3 - "STATIC GUI Operation" on page 19.), and message meaning (See CHAPTER 4 - "Messages" on page 93.).
- If you have our *CAPBAK* tool, use the supplied `static.ksv` file to watch the session run (see below for instructions).

  To use the supplied `static.ksv` file, initialize two xterm-type windows by using the mouse to click on **New Windows** or issuing the command `xterm &` from an existing window. Use the mouse to move one to the upper left corner and the other to lower left corner (as shown on the following page).

  Then type the command:

  ```
  Xplabak -S -k static.ksv
  ```

  in the lower left xterm window. This command will issue a call to *STATIC* to playback the same 7 steps you went through. While **Xplabak** is playing back the session, do not interrupt the keyboard and mouse input. Playback is done when you see the message, `"Playback complete."` appearing on the lower left window.

  When using the supplied `static.ksv` file to playback a *STATIC* session, your display should look like the figure (See Figure 9 "static.ksv Setup" on page 17.).

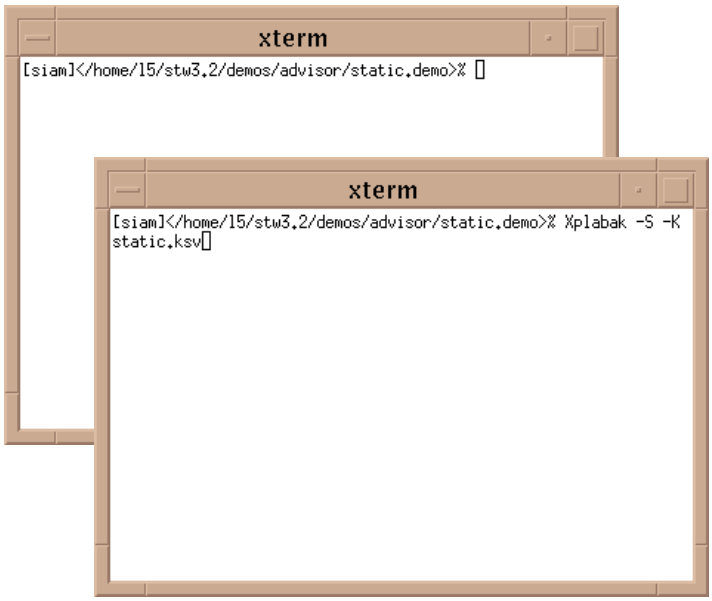

FIGURE 9 static.ksv Setup

CHAPTER 3

STATIC GUI Operation

This chapter covers the basic X Window System graphical user interface operations of *STATIC*. It demonstrates how to load a file(s), analyze the generated report, modify options, and how to customize your reports.
If you are an advanced *STATIC* user, you may just want to refer to the appropriate section for option information (See Section 3.7 - “Modifying the Report Options” on page 32.).

3.1 Using This Chapter

Use this chapter to look up operational questions about *STATIC* you may have. To analyze the report error messages, please refer to the appropriate chapter (See CHAPTER 4 - "Messages” on page 93.).

Turn to the appropriate section for a discussion of the basics of the *STATIC* graphical user interface (See Section 3.2 - “User Interface” on page 19.). If you are already familiar with the OSF / Motif GUI, you may go on to the proper section (See Section 3.3 - “Invoking STATIC” on page 25.).

3.2 User Interface

If you are familiar with the OSF / Motif style graphical user interface, you can go on to the next section (See Section 3.3 - “Invoking STATIC” on page 25.). This section demonstrates using file selection dialog boxes, help menus, message dialog boxes, option menus, and pull-down menus.

File Selection Box

You must use the file selection box to select the file(s) you want *STATIC* to analyze. Refer to the next figure for each of the dialog box’s components (See Figure 10 "Using a File Selection Dialog Box" on page 21.).

Filter entry box Specifies a directory mask. When you click the **Filter** push button, the directory mask is used to filter files or directories that match this mask (or pattern).

Directories Lists directories in path defined in the **Filter** entry box.

Files Lists files in path defined in the **Filter** entry box.

Scroll Bars Move up / down and side / side in the **Directories** and **Files** selection windows. You use them to search for the appropriate directory or file.

Selection entry box

Selects and enters file name.

Use the three push buttons at the bottom of the dialog box to issue commands:

OK Accepts the file in the **Selection** entry box as the new file or the file to be opened and then exits the dialog box.

Filter Applies the pattern you specified in the **Filter** entry box. It lists the directories and files that match that pattern.

Cancel Cancels any selections made and then exits the dialog box. No file is selected as a result.

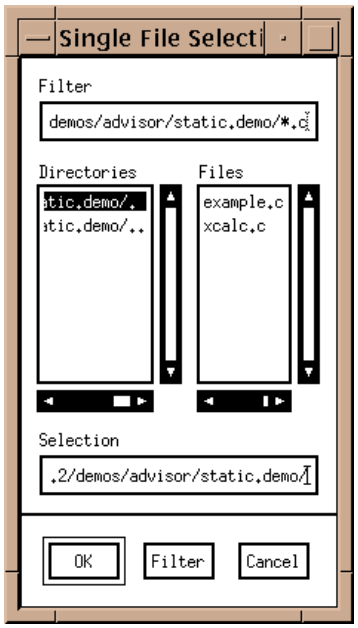

FIGURE 10 Using a File Selection Dialog Box

To use a file selection dialog box, follow these steps:

1. You can restrict the file selection operation to a named region (directory path) by typing in a directory path name in the **Filter** entry box or by clicking on a path name in the **Directories** selection window. Then click on the **Filter** push button.
2. Select a file by clicking on an already existing source file you want *STATIC* to process in the **Files** selection window or type in the file name in the **Selection** entry box, with no limit on character length.
3. To select a source file name, do one of these three things:
   - Double click on the file in the **File** selection window,
   - Highlight the file in the **File** selection window or type in the file name in the **Selection** entry box and click **OK**, or
   - Highlight or type in the file name and press the **<ENTER>** key.

Help Boxes

STATIC provides on-line help. This on-line help will automatically bring up the text corresponding to the window from which you invoke it. In other words, if you invoke it at the **Options** pull-down window's **Error** window, the **Help** window will automatically display information pertinent to the **Error** window. Here's how to use a help frame:

1. Once it is invoked, the text should correspond to the window from which you invoke it.
2. You can use the scroll bars to move up / down and side / side.
3. If you don't see what you need, you can search for specific text. To do this:
   - Click on the **Action** pull-down menu and select **Search**.

     A dialog box (shown below) pops up.
   - Type in the pattern you want to search for and then click on **OK** or press the **<ENTER>** key.

     If the pattern is found, the help frame will automatically scroll to the location of the pattern.
4. If you select another **Help** option from another window, while the current one is displayed, the **Help** window will automatically scroll to the context of the new window.
5. To exit, click on **Quit**.

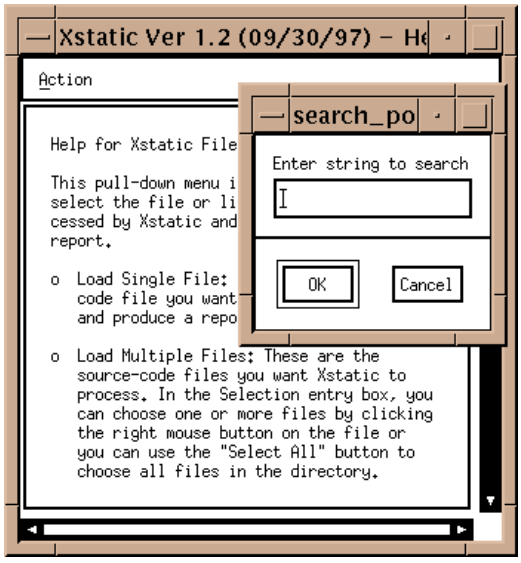

FIGURE 11 Using the Help Dialog Box

Message Boxes

Pop-up message dialog boxes have three purposes:

1. They display warnings and error information.
2. They ask you to verify that you want to perform a task.
3. They ask you to enter a command.

To remove a message box after you have read it or to tell *STATIC* to go ahead with a command, click the **OK** push button. If you want to cancel a command, click the **Cancel** push button.

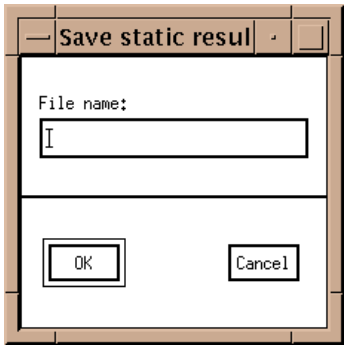

FIGURE 12 Using a Dialog Box

Pull-Down Menus

Pull-down menus are located within the menu bar. They often contain several options. To use pull-down menus and their options, follow these steps:

1. Move the mouse pointer to the menu bar and over the menu containing the item.
2. Hold the left mouse button down. This displays the items on the menu.
3. While holding down the left mouse button, slide the mouse pointer to the menu item you want to select. The menu item is highlighted in reverse shadow.

   Three dots at the right of the menu item indicates that selecting the item will bring up a pop-up window.

   An arrow to the right of the menu item indicates that the item is a submenu (or cascading menu).

   To display the submenu, slide the mouse pointer over the arrow. You can then select an item on the submenu.
4. Release the mouse button while the desired item is highlighted to activate the command. To exit the function without selecting anything, simply drag the mouse pointer off the menu before releasing the mouse button to not activate anything.

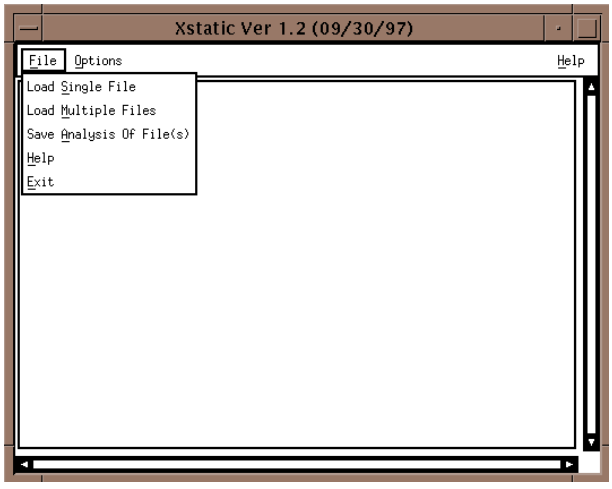

FIGURE 13 Using a Pull-down Menu

3.3 Invoking *STATIC*

To start *STATIC* from your working directory, type this command:

```
Xstatic
```

The **Main** window will pop up. A configuration file `static.rc` is automatically loaded. It contains the default settings for the available error messages, available flags, library header, size, compiler and strong type options. These options, when changed, can significantly change the behavior of *STATIC*. After experimenting with running *STATIC* and modifying the options (See Section 3.7 - "Modifying the Report Options" on page 32.) (See Section 3.8 - "Saving Modifications" on page 87.) (See Section 3.9 - "Customizing STATIC" on page 90.), you may want to permanently change these default settings (See Section 3.9 - "Customizing STATIC" on page 90.).

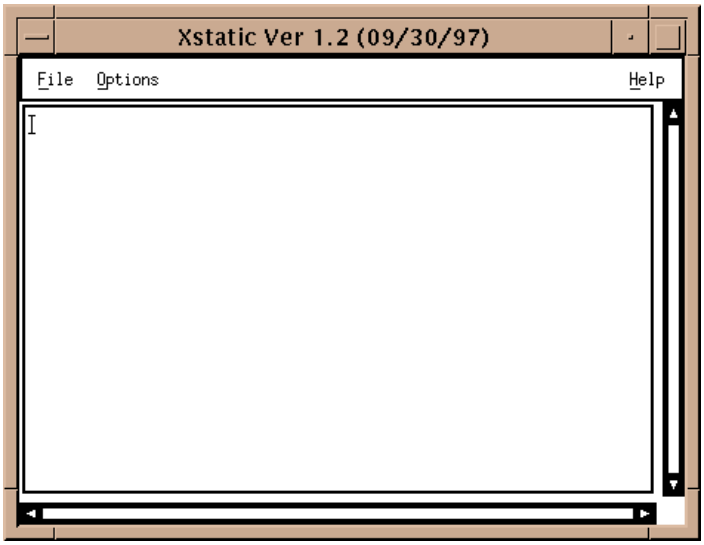

FIGURE 14 Invoking the Main Window

If you have the **STW** product tool set, you can invoke *STATIC* by typing the command:

```
stw
```

1. The **STW** window (shown below) pops up.
2. Click on the **Advisor** activation button.
3. The **STW/Advisor** window pops up.

4. Click on *STATIC*. The *STATIC* window pops up.

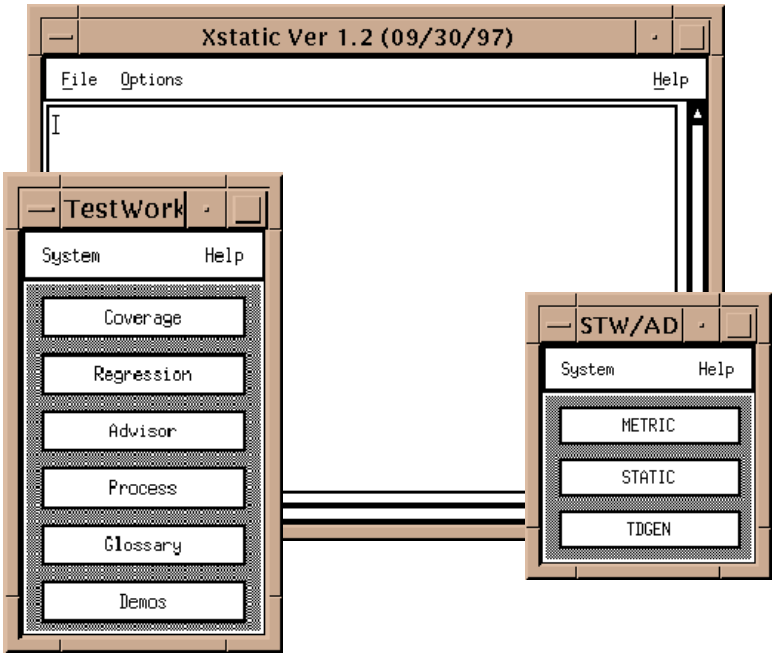

FIGURE 15 Invoking *STATIC* from the STW Suite

3.4 Processing Source Code

Because *STATIC* is a static code analyzer, you do not have to do anything special to the code. To use *STATIC*, all you have to do is select a source code file name, and processing is automatic.

STATIC will automatically generate a report that contains messages regarding your code. These messages indicate areas of your code that may have errors. There are a total of 950 messages possible. Because certain messages may not indicate programming errors for your application, you can turn off particular messages.

3.5 Selecting a Source Code File

Files can be C source files (or *modules*).

Here's how to select a source code file:

1. Click on the File pull-down menu.
2. Select the **Load Single File** option. The file selection dialog box below pops up.

   For further information on using the file selection dialog box, please refer to the appropriate section (See Section 3.2 - "User Interface" on page 19.).
3. Select a source code file.
4. When *STATIC* has processed the source code file, it will automatically create a report. This report is shown in the display area of the **Main** window.

3.5.1 Selecting Multiple Source Code Files

STATIC also allows you to select more than one file for analysis. Here's how to select multiple source code files:

1. Click on the **File** pull-down menu.
2. Select the **Load Multiple Files** option. The file selection dialog box below pops up.

   For further information on using the file selection dialog box, please refer to the appropriate section (See Section 3.2 - "User Interface" on page 19.).
3. To select more than one file, do one of two things:
   - Highlight the files in the **File** selection window by clicking on the actual file names.
   - You can select all of the files, by clicking on the **Select All** button.
4. Click on **OK**.
5. When it has processed the source code files, it will automatically create the report.

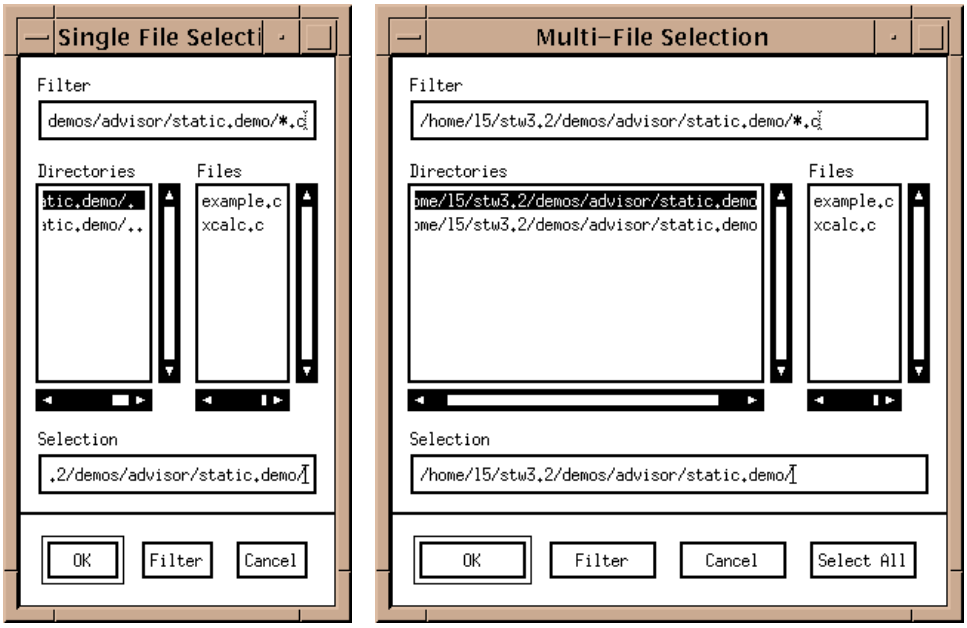

FIGURE 16 Selecting Single or Multiple Source Code File(s)

3.6 Analyzing the Report

After a source code file or multiple files have been loaded into *STATIC*, a report similar to the one below is generated. You can use the scroll bars to move up and down and side to side. It contains the following information:

1. Module Name - The path and the source code file name.
2. Program Statement.
3. Line Number. For each message, the source code file line number that it applies to is indicated.
4. The Type of Message. The type of message for the program statement is indicated. *STATIC* reports on the following types of messages: Syntactical, Internal, Fatal, Warning, Informational, and Elective Notes.
5. Number and Message. Each message type is identified by a number and brief description. Please refer to the correct chapter for further message information (See CHAPTER 4 - "Messages" on page 93.).

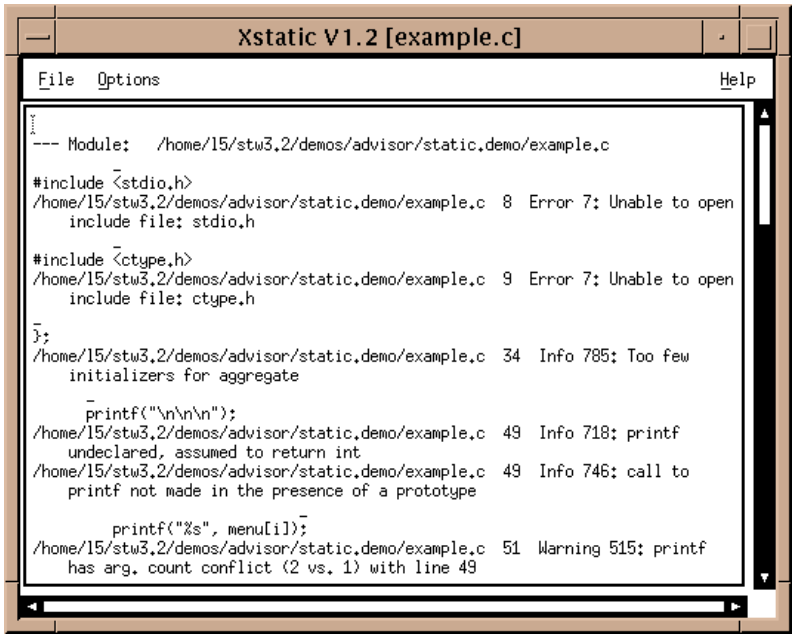

FIGURE 17 Report

3.6.1 Writing the Report to a File

After a report is generated, you may want to save it to a file. If you choose not to do this, your report will not be saved.

To save the report:

1. Click on the **File** pull-down menu.
2. Select **Save Analysis of File(s) option**.
3. The **Save static results as** window (shown below) pops up.
4. Click the mouse pointer in the specification region. When a cursor appears, type in the name of the file you want the report saved to.
5. Click on **OK**.

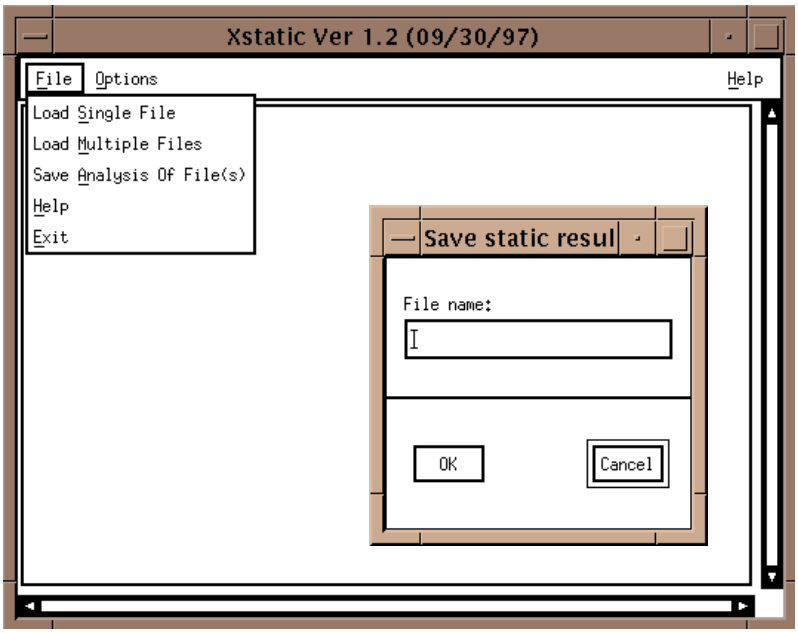

FIGURE 18 Saving the Report

3.7 Modifying the Report Options

By now you probably have seen *STATIC*'s power. It will find things in your code that you probably never would have realized. After a few runs, you may notice that some of the messages are unnecessary for your application. *STATIC* allows you to suppress or to activate messages as well as turn on available flags and options.

To activate or suppress these options, you can do one of two things:

1. Modify the options through the GUI.
2. Manually edit the `static.rc` configuration file. This file includes several other files which list the default options. Each file represents a category of options. `static.err`, for instance, represents the default error options.

When you are familiar with *STATIC* and its options and are ready to customize it to your own options, you can do most of your editing in the configuration files and then make minor modifications by using the GUI. Minor modifications can then be saved to the default configuration files when you exit *STATIC*. See the correct section for further information (See Section 3.10 - “Exiting STATIC” on page 91.).

This Section describes how to modify options through the GUI and details the available options.

3.7.1 Error Messages Options

Because the report is substantial, you may find it beneficial to suppress certain messages.

Most message are defaulted on, except special elective notes. These messages are listed in the 900 to 950 range. Please refer to the correct chapter to see if you want any of these message turned on (See CHAPTER 4 - "Messages” on page 93.).

To turn on or turn off messages:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu and select **Error**.
3. The **Error Options** window pops up. You can use the scroll bars in the **Error Option Set** window to see which messages are already suppressed.

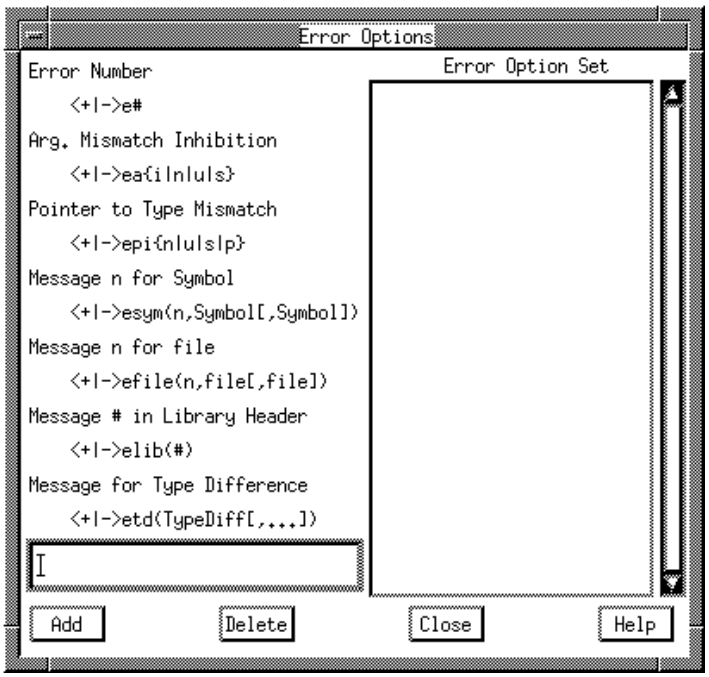

FIGURE 19 Error Options Window

4. The **Error Options** window's **Add** and **Delete** buttons allow you to add or delete inhibition/enabling options to the **Error Option Set**.

   To add a switch to the **Error Option Set** window:

   - Position the mouse pointer so it is in the specification region and click. A cursor should appear.
   - Type in the option you would like to add, such as `-e720`.
   - If you want the option listed at the bottom of the **Error Option Set**, click on **Add**.
   - If you want the option listed at a specific location in the **Error Option Set** window, highlight the option where you would like the new option to go below and then click on **Add**. The new option will be inserted below the option you highlighted.

   This is recommended if you are planning on having a lot of switches. By placing all the `-e#` options together, for instance, it's easier to figure out which messages are suppressed or enabled.

To delete a switch in the **Error Option Set**:

- Highlight the switch you would like to remove.
- Click on **Delete**.
- The option should be removed.

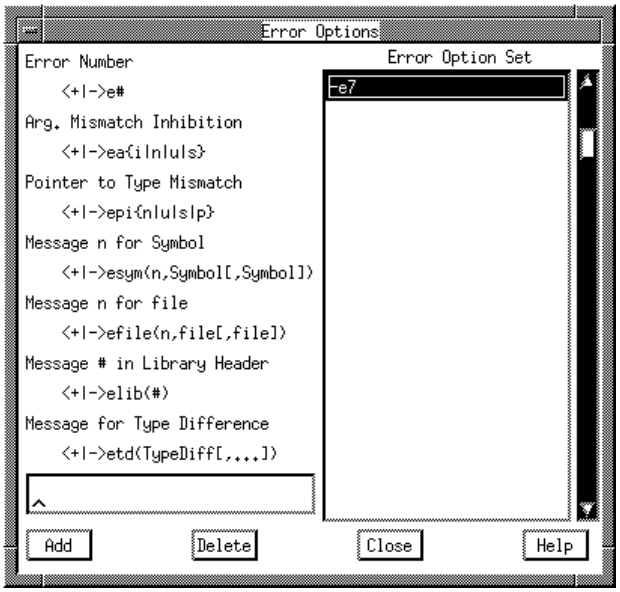

FIGURE 20 Grouping Messages Together

5. Options begin with a plus (+) or minus (-) sign. Options beginning with - inhibit error messages; options beginning with + turns on a message. Because only the 900 level messages are defaulted off, you will only use the + with these messages.

These are the available options to suppress or restore messages:

-e#	Inhibits
+e#	(# is the number of the numeric pattern) Re-enables error message(s) #. For example `-e504` will turn off error message 504. The number designator may contain the ? wild card character. For example `-e7??` will turn off all 700 level errors (informational messages).
`-ea`*letter*	Inhibits

+ea*letter* Activates.Argument Mismatch Switch. *letter* is one of:

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

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

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

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

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

`eau` Refers to type differences where one type is a `signed` and the other an `unsigned` quantity of the same type.

For example, if the function `f` expects an unsigned integer and `n` is an `int`, then the call, `f(n)`, will normally draw warning message `#516`. This message will be inhibited if `eau` is set.

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

`-ep`*letter* Inhibits.

`+ep`*letter* Activates. Pointer to Type Mismatch Switch. This option refers to pointer-to-pointer mismatch (Error 64) across assignment or implied assignment as in initializers, `return` from function, or passing arguments in the presence of a prototype. By selecting one or more of these options, the user can suppress notification of this error for selected pointer differences. *letter* is one of:

`n`	nominal
`p`	all indirect values
`s`	same size
`u`	unsigned vs. signed
`epn`	The pointed-to types differ nominally. For example pointer to `short` vs. pointer to `int` where `int` and `short` are the same size.
`epp`	The pointed-to types differ in anyway imaginable. Said another way... "Pointers are pointers".
`eps`	The pointed-to types differ but they are the same size. For example, pointer to `long` vs. pointer to a `union` containing a `long` but nothing larger.
`epu`	The pointed-to types differ in that one is an `unsigned` version of the other. For example, pointer to `char` being assigned to pointer to `unsigned char`.

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

-esym (*n*,*Symbol*[,*Symbol*]...) Inhibits

+esym (*n*,*Symbol*[,*Symbol*]...) Re-enables error message *n* for the indicated symbols. This is one of the more useful options because it inhibits messages with laser-like precision. For example -esym(714,alpha,beta) will inhibit error message 714 for symbols alpha and beta. (As in all options, embedded blanks are not permitted). Only messages that are parameterized by the identifier *Symbol* can be so suppressed (See Section 4.2 - "Message Glossary" on page 94.). This error inhibition is independent of the -e# option. For a message regarding a particular symbol to be reported, its number must not be inhibited by -e# and it must not be inhibited by -esym(*n*,*Symbol*). For example, the combination:

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

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

-efile (*n*,*file*[,*file*]...) Inhibits

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

-elib # Inhibits

+elib # Re-enables error message # in library headers. 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. For example, `-elib(7??)` will inhibit informationals within library headers.

`-etd` (*TypeDiff*[,...]) Inhibits

`+etd` (*TypeDiff*[,...]) Re-enables messages arising through certain specified type differences. The chapter details the various type differences (under the heading *TypeDiff*) and some messages are parameterized by type differences (See Section 4.2 - “Message Glossary” on page 94.).

For example, `-etd(ellipsis)` will inhibit messages reported as the result of two function types differing in that one is specified with an ellipsis and the other is not. The *TypeDiff* must be an identifier or of the form *identifier/identifier*; it may not be of the form *Type* = *Type*, or *Type* vs. *Type* or otherwise compound.

Examples of Error Inhibition Options

-e720

will inhibit message 720.

+e9??

will turn on all the 900 level messages.

-e??? +e526

will turn off all messages except number 526.

-epp -eau -esym(526,alpha)

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

3.7.2 Flag Options

STATIC allows you to turn on flags. These flags give directives to *STATIC* on how to treat data types and syntax structure.

To turn on any flags:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu and select **Flag**.
3. The **Flag Options** window pops up. It lists all the flags that can effect how *STATIC* treats data types and syntax structure.
4. To select an option, simply click on the corresponding check button.

FIGURE 21 Flag Options Window

The **Flag Options** window consists of the following options:

- `Abbreviated Structure (fab)` **-** If this flag is ON, structure references may be abbreviated. Thus, instead of `s.a.b`, if it would cause no ambiguity, you may use `s.b`. Few compilers support this feature.

- `Anonymous Union (fan)` - If this flag is ON, anonymous unions are supported. Anonymous unions appear within structures and have no name within the structure so that they must be accessed using an abbreviated notation. For example:

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

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

  This is a feature of the Microsoft 6.0 compiler and is also in C++.

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

- `Char is Unsigned (fcu)` - If this flag is ON, all `char` declarations are assumed to be equivalent to `unsigned char`.

  This is useful for compilers which, by default, treat `char`s as `unsigned`. Note that this treatment is specifically allowed by the ANSI standard. That is, whether `char` is `unsigned` or `signed` is up to the implementation. See also the `String Unsigned` flag in subsequent explanation of #define.

- `Directory of Including File (fdi)` - 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 running *STATIC* on a file in some other directory as in:

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.

- `Pointer Difference is Long (fdl)` - This flag specifies that the difference between two pointers is typed `long`. Otherwise the difference is typed `int`.

  This flag is automatically adjusted upon encountering a `typedef` for `ptrdiff_t`.
- `Deduce Return Mode (fdr)` - The return mode of a function has to do with whether the function does, or does not, return a value. This flag only affects function definitions and declarations that do not have an explicit return type. This can be a very valuable option for older C programs. If the flag is ON, `return` statements are examined to determine the return mode of such a function. If the flag is OFF, such a function is assumed to return an `int`. With the flag OFF we are adhering strictly to ANSI.
- `Float to Double (ffd)` - If this flag is ON `float` expressions are automatically promoted to `double` when being used in an arithmetic expression (just as `char` is promoted to `int`). Automatic `float` promotion is K&R C but not ANSI C.
- `Flush Output Files (ffo)` - When ON, the `fflush()` function is called after each message. Otherwise messages are buffered. If there are many messages it is slightly faster to buffer.
- `Hierarchy Graphics (fhg)` - If this flag is ON, the IBM graphics characters are used to display a type hierarchy tree (See Section 3.7.7 - "Strong Typing Options" on page 61.).
- `Hierarchy of Strong Types (fhs)` - If this flag is ON, strong types are considered to form a hierarchy based on `typedef` statements (See Section 3.7.7 - "Strong Typing Options" on page 61.).
- `Hierarchy of Strong Indexes (fhx)` - If this flag is ON, strong index types are related via the type hierarchy (See Section 3.7.2 - "Flag Options" on page 38.).
- `Integer Model for Enum (fie)` - If this flag is ON, a loose model for enumerations is used. specifically, enumerations are regarded semantically as integers. By default, a strict model is used wherein variables of some enumerated type must be assigned compatible enumerated values 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.

- `Indentation Check on Labels (fil)` - Normally no indentation check is done on labels because frequently they are positioned far to the left of a listing in a position of prominence and easy visibility. If you want labels checked, turn this flag ON. See the correct section for indentation checking (See Section 7.3 - "Indentation Checking" on page 168.).
- `Integral Constants Are Signed (fis)` - If this flag is ON integer constants are typed `int` or `long`, not `unsigned` or `unsigned long`. For example, by the rules of ANSI (See Section 10.1 - "Size of Scalars" on page 197.), `0xFFFF` is considered `unsigned` if `int`s are 16 bits. However, some older compilers regard all integral constants as signed. To mimic these use `+fis`.
- `K&R Preprocessor (fkp)` - A number of preprocessor facilities are allowed by the ANSI C Standard which are not allowed in K&R C. These include blanks and tabs preceding the initial # sign. Setting this flag causes strict adherence to the K&R preprocessor specification.
- `Library (flb)` - The option has been made into a flag to permit it to be turned ON and OFF in a recursive setting as when `include` files are being processed. In this way all entries in a particular `include` file can be designated as library entries. This flag has been largely superseded by the notion of library header files (See Section 3.7.3 - "Library Header File Options" on page 47.).
- `Multiple Definitions (fmd)` - Some compilers allow multiple definitions of data items provided they are not accompanied by an initializer. These are referred to in ANSI C as tentative definitions. For example, in the sequence:

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

  some compilers would consider only the last declaration as erroneous. If the `fmd` flag is ON, only the last declaration draws the previously defined error message (number 14). Multiple definitions of functions are always reported.
- `Nested Comments (fnc)` - If this flag is ON, comments may be nested. This allows *STATIC* 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.

- `Output Declared Objects (fod)` - This flag has an effect only when a Lint Object Module is being produced. See option `-oo` (See Section 3.7.8 - "Other Options" on page 78.) Normally, objects declared but not referenced are not placed in the output. With this flag ON, all objects declared are placed there. This has the disadvantage of making the object modules much larger than they need to be. It has the advantage that all declared objects will be cross-checked.
- `Output Library Objects (fol)` - This flag has an effect only when a Lint Object Module is being produced. See option `-oo`. (See Section 3.7.8 - "Other Options" on page 78.)Normally, objects declared when the library flag is set (see `+flb` and/or `-library`) (See Section 3.7.8 - "Other Options" on page 78.) are not placed in the output. With this flag ON, all library objects are placed in the output module. It is not usually necessary to set this flag ON when creating a Lint Object Module that describes a library (See Section 3.2 - "User Interface" on page 19.)..
- `Pointer Casts Retain lvalue (fpc)` - This flag can be used to legitimize a non-ANSI non-K&R practice which is rife in the C community. For example if you wanted to add 1 (1 byte not 1 `int`) to an `int` pointer (`pi`) then you could write:

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

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

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

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

- `Precision Limited to Max. of Arg. (fpm)` - This is used to suppress certain kinds of Loss of Precision messages (#734). In particular, if multiplication or left shifting is used in an expression involving `char` (or `short` where `short` is smaller than `int`) an unwanted loss of precision message may occur. For example, if `ch` is a `char` then:

```
ch = ch * ch
```

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

For example

ch <<= 1

is not greeted with Message 734.

- `Parameters Within Strings (fps)` - This flag, when set ON, allows macro parameters to be substituted within strings as in:

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

which prints both the name and the value of the parameter passed to the macro `printi`. This depends on the substitution of a macro parameter within a string constant and is supported by many compilers but is now expressly forbidden by ANSI C. There are other ways to accomplish this task provided your compiler supports them (See Section 9.3.8 - “# Stringize operator” on page 194.). If it doesn’t, set this flag ON; see also Warning #607 in the chapter (See CHAPTER 4 - "Messages” on page 93.).

- `Read Binary (frb)` - When this flag is ON, all files `fopen`ed on input are given a mode of `rb` rather than `r`. This is to resolve an obscure problem that can arise with some editor / compiler combinations. On a system (such as MS-DOS) that uses **CR-LF** to separate lines, some editors do not insert a **CR** between lines (to save space) and some run-time libraries will not stop (with `fgets`) on just an **LF** unless read with `rb`. Using this option will handle the situation.
- `Structure Assignment (fsa)` - If this flag is ON, structure assignment is assumed to be valid. Functions, actual arguments and parameters may be typed `struct` or `union` and such objects are allowed to be used in assignment.
- `String Unsigned (fsu)` - With this flag ON, a string of constant characters (as in “...”) is regarded as a pointer to an unsigned character. See also the `fcu` flag in subsequent explanation for FZu..
- `Unsigned Long (ful)` - If the unsigned long flag is ON, then `unsigned long` is a valid type.
- `Variable Arguments (fva)` - Functions declared or defined while this flag is ON are assumed to have a variable argument list. Warning messages (515 and 516) reporting inconsistencies between argument lists are suppressed for such functions. For example:

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

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

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

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

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

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

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

- `Void Data Types (fvo)` – If this flag is ON, `void` is recognized as a type and functions declared as `void` are assumed to return no value.
- `Varying Return Mode (fvr)` – The return mode has to do with whether particular functions do, or do not, return a value. If this flag is ON when a function is defined or declared, then the function does not have to be consistent in this respect. Error messages arising out of an incompatibility between calls to the function and the function declaration or between two calls or between return statements and either of the above are inhibited. For example, since `strcpy()` returns a string (in most standard libraries) and since the string is seldom used, it would be wise to set this flag ON for at least one of the declarations of `strcpy()`.

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

`-esym(534,`*name1*`,`*name2*`, ...)`

- `Exact Array (fxa)` – This flag, if ON, selectively inhibits promotion of array arguments and array parameters (for the purpose of type matching) to pointers. This provides a more strict type-checking in function calls than is normally obtainable. In particular, only arrays may be passed to parameters declared as array and the sizes, if specified, must match. On the other hand, both arrays and pointers may be passed to a parameter typed as

pointer. See the correct section for exact parameter matching information (See Section 7.7 - “Exact Parameter Matching” on page 174.).

- `Exact Char (fxc)` – This flag, if ON, inhibits promotion of `char` or `unsigned char` arguments and parameters (for the purpose of type matching). Normally these types are silently promoted for argument passing to `int`, and this promotion can hide unintended disagreements between parameter and argument. See the correct section for exact parameter matching information (See Section 7.7 - “Exact Parameter Matching” on page 174.).
- `Exact Float (fxf)` – This flag, if ON, inhibits promotion of `float` arguments and parameters (for the purpose of type matching). Normally these types are silently promoted to `double`, and this promotion can hide unintended disagreements between parameter and argument. Seethe correct section for exact parameter matching information (See Section 7.7 - “Exact Parameter Matching” on page 174.).
- `Exact Short (fxs)` – This flag, if ON, inhibits promotion of `short` and `unsigned short` arguments and parameters (for the purpose of type matching). Normally these types are silently promoted to `int`, and this promotion can hide unintended disagreements between parameter and argument. See the correct section for exact parameter matching (See Section 7.7 - “Exact Parameter Matching” on page 174.).
- `Sizeof is Long (fzl)` – If this flag is ON, `sizeof()` is assumed to be a `long` (or `unsigned long` if `-fzu` is also ON). The flag is OFF by default because `sizeof` is normally typed `int`. This flag is automatically adjusted upon encountering a `size_t` type. This flag is useful on architectures where `int`s are not the same size as `long`s.
- `Sizeof is Unsigned (fzu)` – If this flag is ON, `sizeof()` is assumed to return an unsigned quantity (`unsigned long` if `-fzl` is also ON). This flag is automatically adjusted upon encountering a `size_t` type.

3.7.3 Library Header File Options

Note: This section is *not* about how to include header files that may be in some directory other than the current directory. For that information, see the correct section (See Section 9.2 - "include Processing" on page 192.). This section explains how information about libraries is passed to *STATIC*. This usually, but not always, takes the form of header files.

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 *STATIC* is concerned, are:

1. The source code may not be available for *STATIC*.
2. The library is used by programs other than the one you are running *STATIC* on.

Information about libraries is conveyed to *STATIC* via Library Headers. 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

```
#include <stdio.h>

main()
    {
    printf( "hello world" );
    }
```

Without the header file, *STATIC* 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. 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).

With the inclusion of `stdio.h` (assuming `stdio.h` contains a declaration for `printf`), *STATIC* will no longer 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), *STATIC* will understand that source code for `printf` is not necessarily available, see clause (1) on the previous page, and will not issue warning 526 either.

Note: Other messages associated with library headers are not suppressed automatically. But you may use `-elib` for this purpose. See the correct section for error inhibition options (See Section 3.7.3 - “Library Header File Options” on page 47.).

Because of clause (2), not all components of a library header file need to be fully utilized over the course of compiling a program. Such components include: declared data objects and functions, types specified with typedef, macros specified with #define, and struct, union and enum declarations and their members. For these components, messages 749-770 are suppressed. See the correct section for weak definial information (See Section 7.8 - “Weak Definials” on page 176.).

A header file can become a library header file if:

1. It falls within one of the four broad categories of the option `+libclass`, viz. `all`, `ansi`, `angle` and `foreign` (described below), and not excluded by either the `-libdir` or the `-libh` option.
2. OR, for finer control, it comes from a directory specified with `+libdir` and is not specifically excluded with -libh.
3. OR, for the finest control, is specifically included by name vi +libh.
4. OR, is included within a library header file.

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.

To specify if or when a header is a library header file:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu and select Library.
3. The **Library Options** window pops up.

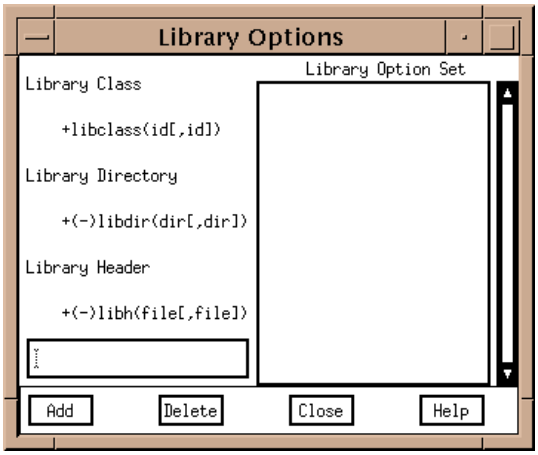

FIGURE 22 Library Options Window

4. The **Library Options** window's **Add** and **Delete** buttons allow you to add or delete library header options. To add a library option:
   - Position the mouse pointer so it is in the specification region and click. A cursor should appear.
   - Type in the option you would like to add.
   - If you want the option listed at the bottom of the **Library Option Set**, click on **Add**.
   - If you want the option listed at a specific location in the **Library Option Set** window, highlight the option where you would like the new option to go below and then click on **Add**. The new option will be inserted below the option you highlighted.

To delete a switch in the **Library Option Set**:

- Highlight the switch you would like to remove.
- Click on **Delete**.
- The option should be removed.

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 other than the current directory.

Note: 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)
```

(This should not be construed as an endorsement for using full path names in #include files.) "

identifier option, ansi

ansi The 'standard' ANSI header files, viz.

assert.h	locale.hstddef.h
ctype.h	math.hstdio.h
errno.h	setjmp.hstdlib.h
float.h	signal.hstring.h
limits.h	stdarg.htime.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(/compiler)
```

+libh(os.h)

requests that no header files be regarded as library files except those coming from directory `/compiler` and the header `os.h` from whatever directory. 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.

Note: A file specified as #include "/compiler/i.h" is not regarded as libdir(/compiler). Only files found in /compiler via -i searching are so regarded.

+libh(*file[, file]...*) Adds

-libh(*file[, file]...*) Removes files from the set that would otherwise be determined from the +libclass option.

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. Note that the **libh** option is cumulative whereas the **libclass** option overrides any previous **libclass** option, including the default.

3.7.4 Size Options

This size options allow you to set the sizes of various scalars (shorts, `int`s, etc.) for the target machine. The separate setting of these parameters is not normally necessary as the default settings are consistent with most compilers in your environment. Use the size options for specifying architectures other than the native architectures.

To change sizes of scalars:

1. Click on the **Options** pull-down menu.

2. Drag the mouse to the **Modify** submenu and select Size.
3. The **Size Options** window pops up. It lists all the size options on the left side and the default scalars sizes on the right side.
4. To edit a size, simply click on the corresponding specification region. When the cursor appears, you can begin editing.
5. If you want to keep your changes, click on the **Apply** button. If not, click on the **Reset** button.

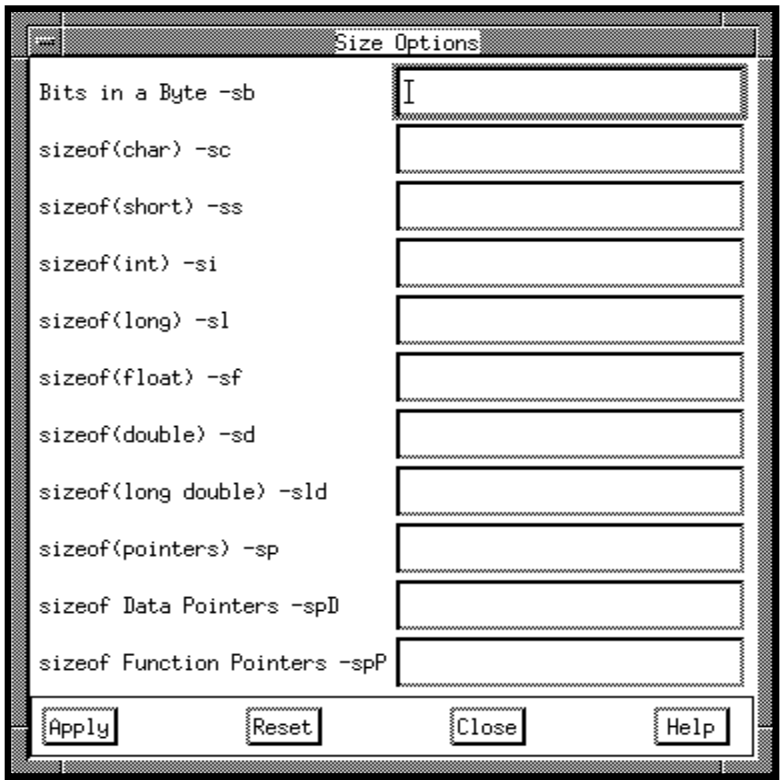

FIGURE 23 Size Options Window

For example, a

`sizeof(int) -si size of 2`

specifies that the size of integers is two bytes. In the list below # stands for a small integer.

Bits in a Byte -sb

The number of bits in a byte is #.8 is the default. The number of bits in an `int` is presumed to be sizeof(int)

times this quantity. The maximum integer is determined from this quantity by assuming a 2's complement machine. The maximum integer, in turn, is used to determine whether a constant is `int` or long.

sizeof(char) -sc

The default value is, of course, 1 (this option is present for completeness. Do you really want to set the size of a `char` to something other than 1?)

sizeof(short) -ss

`sizeof(short)` becomes #. 2 is the default.

sizeof(int) -si

`sizeof(int)` becomes #. 4 is the default.

sizeof(long) -sl

`sizeof(long)` becomes #. 8 is the default.

sizeof(float) -sf

`sizeof(float)` becomes #. 4 is the default.

sizeof(double) -sd

`sizeof(double)` becomes #. 8 is the default.

sizeof(long double) -sld

`sizeof(long double)` becomes #. 8 is the default.

sizeof(pointers) -sp

sizeof pointers becomes #. This option sets both program and data pointer sizes to the same value. 8 is the default. This option sets both program and data pointer sizes to the same value.

sizeof Data Pointers -spD

Indicates the size of data pointers is # bytes. This has no effect on the assumed size of program (function) pointers. The default is 8.

sizeof Function Pointers -spP

Indicates that the size of a Program (function) pointer is # bytes. This has no effect on the assumed size of data pointers. The default is 8.

3.7.5 Compiler Vendor Options

All compilers are slightly different owing largely to differences in libraries and preprocessor variables, if not actually to differences in the language processed. The key to coping with these differences is the selection of *STATIC*'s vendor switches or compiler options.

To select the compiler vendor:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu.
3. Drag the mouse to the **Compiler** submenu and select **Vendors.**
4. The **Compiler Options** window pops up. It lists all the compiler vendors. If your compiler is not found in the group above, you may want to modify `co.lnt` which is the generic compiler options file. In addition, if your compiler does not provide prototypes and is not in the list above, you may wish to modify the file `sl.c` which is a generic standard library file.
5. To select an option, simply click on the corresponding check button.
6. Click on **OK**.

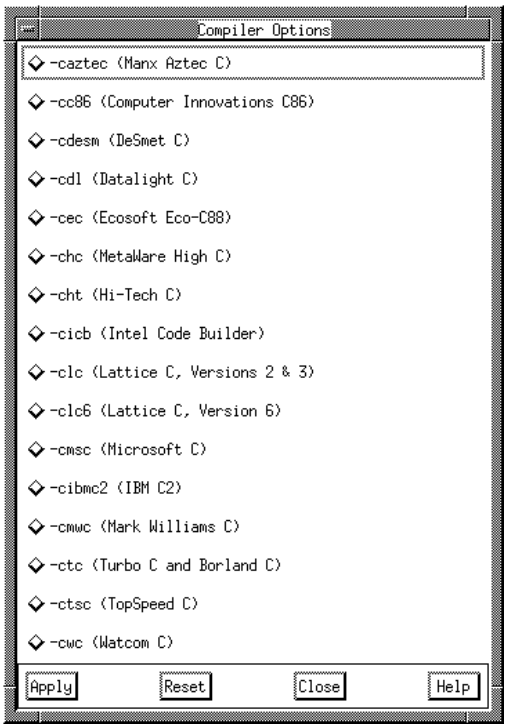

FIGURE 24 Compiler Options Window

The **Compiler Options** window consists of the following options:

-caztec (Manx Aztec C compiler). Enables **__FUNC__** and **MPU8086** and, for large data pointers, **__LDATA**.

-cc86 (Computer Innovations C86). Enables **_C86_BIG** for large data pointers.

-cdesm (DeSmet C). Enables the symbol **LARGE_CASE** for large data pointers.

-cdl (Datalight C). Enables **DLC**, **I8086** and one of **I8086L**, **I8086D**, **I8086P** and **I8086S** depending on memory model. Also LPTR is defined to be 1 for large data models and SPTR is defined to be 1 for small data models.

-cec (Ecosoft Eco-C88). Enables **__ECO**, and, for memory models having large data pointers, enables

__BIGDATA, and, for memory models having large code pointers, enables **__BIGCODE**.

-chc (MetaWare High C). Pre-assigns 0 to **_stdio_defs_included**, **_1167**, **_HIGHC_**.

-cht (Hi-Tech C). Pre-assigns 0 to symbols **z80** and **m68k** and pre-assigns 1 to symbol **i8086**.

-cicb (Intel Code Builder). Pre-assigns variables **LINT_ARGS**, **_TIMESTAMP_**, **CH_TIME**, **_INTELC32_**, **_ARCHITECTURE_**, **386** and sets the size of **int**s and **pointer**s to 4 bytes and allows **'$'** in identifiers.

-clc (Lattice C, Versions 2 & 3). **MSDOS** is enabled. **SPTR** is defined to be 1 if the memory model uses small data pointers and 0 otherwise. LPTR is defined to be 1 if the memory model uses large data pointers and 0 otherwise. In addition, **CPM80**, **CPM86 CPM68**, **LATTICE**, and **I8086** are enabled and one of **I8086L, I8086D**, **I8086P** or **I8086S** depending on memory model.

-clc6 (Lattice C, Version 6). This is like **-clc** except that, in addition, preprocessor words ANSI, NULL and LC60 are enabled and CPM* are not. The user should enable DOS, FAMILY or OS2 if using these symbols.

Also, keywords **align**, **critical**, **noalign**, **nopad**, **pad, private**, **interrupt**, **near**, **far**, **huge**, **pascal**, **actual**, **inline** and their double underscore prefix versions are enabled.

-cibmc2 (IBM C2). Both of these options have the same effect. They enable **MSDOS**, **M_I86** and one of **M_I86LM**, **M_I86CM**, **M_I86MM** and **M_I86SM** depending on memory model. Also, keywords (reserved words) **_loadds**, **_export**, **_saveregs _asm**, **_based**, **_segment**, **_segname** and **_self** are enabled, as well as the Microsoft keywords which are: **near**, **far**, **huge**, **pascal**, **fortran**, **cdecl**, **interrupt**, **_near**, **_far**, **_huge**, **_pascal**, **_fortran**, **_cdecl**, **_interrupt**, **_fastcall**. The **//** form of comment is understood. If **-A** (or **-Za**) is set then **NO_EXT_KEYS** is enabled and then special keywords and comment control are disabled.

-ctc (Turbo C and Borland C). This option supports Turbo C, and the C portion of Turbo C++ and Borland C++. **__TURBOC__**, **__MSDOS__**, **__CDECL__** are defined to be 1 and one of **__LARGE__**, **__COMPACT__**, **__MEDIUM__**, **__SMALL__** are defined (to be 1) according to the memory model selected. Also, **__STDC__** is treated differently than for other compilers. **__STDC__** is by default undefined and is defined (to be 1) only if -A is set.

Additional keywords supported are: **asm**, **_ss**, **_es**, **_ds**, **_cs**, and **_seg** (this is in addition to **far**, **near**, **huge, pascal**, **fortran** and **cdecl** which are enabled by default). For the large model, **sizeof** is assumed to be unsigned long.

Register keywords (as in Turbo C, these are pre-declared to be of type **unsigned** or **unsigned char** depending on whether the associated register is 16 bits or 8 bits long).

_AX _BX _CX _DX _SI _DI _SP _BP

_AH _AL _BH _BL _CH _CL _DH _DL

_DS _ES _RS _SS _FLAGS

-ctsc (TopSpeed C). Enables **M_I86LM**, **M_I86CM**, **M_I86MM**, **M_I86SM** according to memory model (like Microsoft C).

-cwc (Watcom C.) Enables **M_I86** and one o**f M_I86LM**, **M_I86CM**, **M_I86MM** and **M_I86SM** depending on memory model. Enables keywords **__far**, **__near**, **__huge** and **__interrupt**. If **-A** (or **-Za**) is set, then **NO_EXT_KEYS** is enabled. As with all compilers, the Microsoft keywords are enabled by default.

3.7.6 Compiler Customization Options

STATIC allows you to support a number of features in a variety of compilers. With some exceptions, they are used mostly to get *STATIC* to ignore some nonstandard constructs accepted by some compilers. To turn on any of these features:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu.
3. Drag the mouse to the **Compiler** submenu and select **Customizations**.
4. The **Compiler Customization Options** window pops up. When you add more options, you can use the scroll bars in the **Compiler Option Set** window to move up / down or side / side. The default option is **-d_NO_PROTO**.

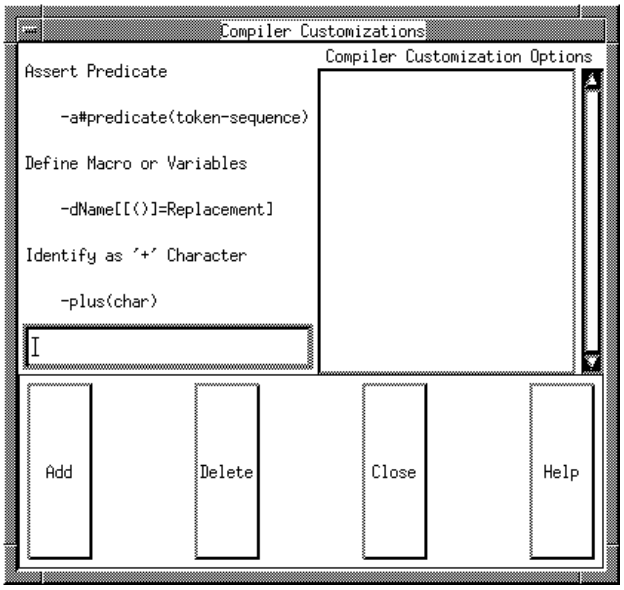

FIGURE 25 Compiler Customization Options Window

5. The window's **Add** and **Delete** buttons allow you to add or delete compiler options to the **Compiler Option Set** list.

To add an option:

- Position the mouse pointer so it is in the specification region and click. A cursor should appear.

- Type in the option you would like to add.
- If you want the option listed at the bottom of the **Compiler Option Set**, click on **Add**.
- If you want the option listed at a specific location in the **Compiler Option Set** window, highlight the option where you would like the new option to go below and then click on **Add**. The new option will be inserted below the option you highlighted.

To delete a switch:

- Highlight the switch you would like to remove.
- Click on **Delete**.
- The option should be removed.

These are the available compiler options:

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

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

-a#machine(pdp11)

makes the predicate **`#machine(pdp11)`** true. See also the appropriate section for non-ANSI preprocessing .

`-d` *Name()= Replacement*

To induce *STATIC* 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 and is hence not as convenient as using this option. For example, if your compiler supports

`char_varying`(*n*)

as a type and you want to get *STATIC* to interpret this as char* you can use

`-dchar_varying()=char*`

As another example, for VAX-11 C,

`-d_align()=`

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

-plus(*char*) Identifies char as an alternate '+' character.

3.7.7 Strong Typing Options

The notion of strong typing is not usually carefully defined. It generally means the kind of type checking that Pascal has that C does not. These include the following:

1. User-defined types match only through the nominal type, not through the underlying type as is done with C.
2. A special Boolean type is supported which must be used where Boolean's are expected. In C, any scalar can be used as a Boolean and any Boolean is typed `int`.
3. The Pascal-equivalent of **`char`** and enum objects are not automatically converted to and from **`int`** as is done in C. Explicit conversion is required.
4. Every array has an expected index type and every subscript must match this type. In C, any integral can be used as a subscript for any array.
5. Pascal has a set facility implemented as a finite number of bits that are either on or off. In C, one uses bit-wise operations on integral quantities to achieve the same effect. C's approach is more flexible but Pascal sets and their members cannot be improperly mixed.

In addition to these static checks, Pascal systems have run-time checks that include subscript bounds and pointer-NIL checks. We do not include these under the notion of Strong Type checking.

In the pages that follow, each of the static type checks enumerated above will be seen to be represented as options for *STATIC*. We describe how a *STATIC*-like utility can superimpose strong typing wholly or partially on a C program through the use of the **typedef** facility and in the judicious selection of appropriate options.

Additional flexibility is obtained by means of a type hierarchy. In a type hierarchy generic uses of a type are distinguished from, but related to, more specific uses of a type.

What Are Strong Types?

Have you ever gone through the trouble of typedef'ing types and then wondered whether it was worth the trouble? It didn't seem like the compiler was checking these types for strict compliance.

Consider the following typical example:

```
typedef int Count;
typedef int Bool;
Count n;
```

```
Bool stop;
        n = stop ; /* mistake but no warn-
ing */
```

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

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

To turn on any of the strong type options:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu and select **Strong Types**.
3. The **Strong Type Options** window pops up. When you add more options, you can use the scroll bars in the **Strong Option Set** window to move up / down or side / side.

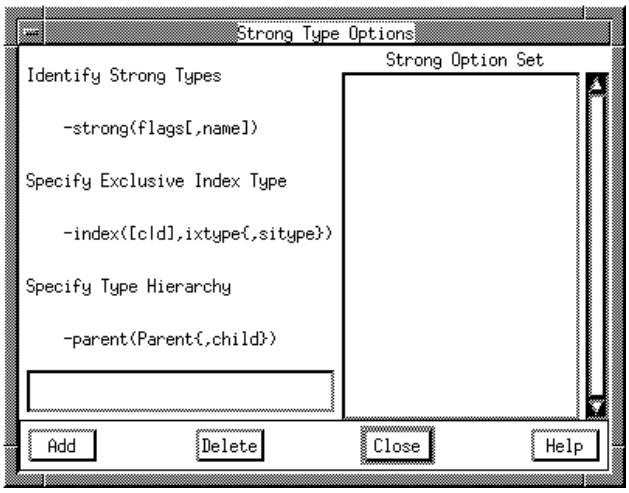

FIGURE 26 Strong Type Options

4. The **Add** and **Delete** buttons allow you to add or delete compiler options to the **Strong Option Set** list.

To add an option:

Position the mouse pointer so it is in the specification region and click. A cursor should appear.

Type in the option you would like to add.

If you want the option listed at the bottom of the **Strong Option Set** window, click on **Add**.

If you want the option listed at a specific location in the **Strong Option Set** window, highlight the option where you would like the new option to go below and then click on **Add**. The new option will be inserted below the option you highlighted.

To delete a switch:

- Highlight the switch you would like to remove.
- Click on **Delete**.
- The option should be removed.

These are the available options:

3.7.7.1 -strong

`-strong`(*flags[, name]...*)

Identifies each name as a strong type with properties specified by flags. Presumably there is a later **`typedef`** defining any such *name* to be a type. This option has no effect on typedef's defined earlier. If *name* is omitted, then flags specifies properties for all **`typedef`**'ed types that are not identified by some other **`-strong`** option.

The *flags* can be:

`A` Issue a warning upon some kind of Assignment to the strong type. (assignment operator, return value, argument passing, initialization). A may be followed by one or more of the following letters which soften the meaning of A.

`i`	Ignore Initialization
`r`	Ignore Return statements
`p`	Ignore argument Passing
`c`	Ignore assignment of Constants

As an example, -strong(Ai,BITS) will issue a warning whenever a value whose type is not BITS is assigned to a variable whose type is BITS except when the variable is being initialized. (If the strong type is a pointer

then &x, where x is a *STATIC* or automatic variable, is considered a constant.)

X Check for strong typing when a value is extracted. This causes a warning to be issued when a strongly typed value is assigned to a variable of some other type (in one of the four ways described above). But note, the softeners (**i**, **r**, **p**, **a**, **c**) cannot be used with **X**.

J Check for strong typing when a value is Joined (i.e., combined) with another type across a binary operator. This can be softened with one or more of the following lower case letters immediately following the **J**:

- **e** Ignore Equality operators (== and !=) and the conditional operator (?:).
- **r** Ignore the four Relational operators (>>= <<=).
- **o** Ignore the Other binary operators which are the five arithmetic operators (+ - * / %) and the three bit-wise operators (| & ^).
- **c** Ignore combining with Constants.

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

B The type is Boolean. Normally only one name would be provided and normally this would be used in conjunction with other flags (if through the fortunes of using a third party library, multiple Booleans are thrust upon you, make sure these are related through a type hierarchy. See Type Hierarchies (See Section 3.7.7 - "Strong Typing Options" on page 61.). The letter 'B' has two effects:

1. Every Boolean operator will be assumed, for the purpose of strong type-checking, to return this type. The Boolean operators are those that indicate true or false and include the four Relational and two Equality operators mentioned above, Unary !, and Binary && and | |.

2. Every context expecting a Boolean, such as an if clause, while clause, second expression of a for statement, operands of Unary ! and Binary || and &&, will expect to see this strong type or a warning will be issued.

b This is like flag **B** except that it has only effect numbered 1 above. It does not have effect 2. Boolean contexts do not require the type.

Flag **B** is quite restrictive insisting as it does that all Boolean contexts require the indicated Boolean type. By contrast, flag **b** is quite permissive. It insists on nothing by itself and serves to identify certain operators as returning a designated Boolean type rather than an `int`. See also the `'l'` flag below.

l Is the Library flag. This designates that the objects of the type may be assigned values from or combined with library functions (or objects) or may be passed as arguments to library functions. The usual scenario is that a library function is prototyped without strong types and the user is passing in strongly typed arguments. Presumably the user has no control over the declarations within a library. Also, this flag is necessary to get built-in predicates such as *isupper* to be accepted with flag B. See example below.

f **f** goes with **B** or **b** and means that bit fields of length one should not be Boolean (otherwise they are). See Bit field example below.

These flags may appear in any order except that softeners for **A** and **J** must immediately follow the letter. There is at most one 'B' or 'b'. If there is an 'f' there should also be a 'B' or 'b'. In general, lower case letters reduce or soften the strictness of the type checking whereas upper case letters add to it. The only exceptions are possibly 'b' and 'f' where it is not clear whether they add or subtract strictness.

If no flags are provided, the type becomes a 'strong type' but engenders no specific checking other than for declarations.

Examples of -strong

For example, the option

-strong(A)

indicates that, by default, all **typedef** types are checked on Assignment (**A**) to see that the value assigned has the same typedef type.

The options:

-strong(A) -strong(Ac,Count)

specify that all typedef types will be checked on Assignment and constants will be allowed to be assigned to variables of type Count.

As another example,

-strong(A) -strong(,Count)

removes strong checking for **Count** but leaves **Assignment** checking in for everything else. The order of the options may be inverted. Thus

-strong(,Count) -strong(A)

is the same as above.

Consider:

```
/-strong(Ab,Bool) */
typedef int Bool;

Bool gt(a,b)
    int a, b;
    {
    if(a) return a > b;  /* OK */
    else return 0;       /* Warning */
    }
```

This identifies **Bool** as a strong type. If the flag **b** were not provided in the **-strong** option, the result of the comparison operator in the first return statement would not have been regarded as matching up with the type of the function. The second return results in a Warning because 0 is not a Bool type. An option of **-strong(Acb,Bool)**, i.e. adding the **c** flag, would suppress this warning.

We do not recommend the option 'c' with a Boolean type. It's better to define

#define False (bool) 0

and

return False;

Had we used an upper case B rather than lower case b as in:

-strong(AB,Bool)

then this would have resulted in a Warning that the if clause is not Boolean (variable a is `int`). Presumably we should write:

```
if( a != 0 ) ...
```

As another example:

```
/*-strong(AJXl,STRING)  */
typedef char *STRING;
STRING s;


  .
    .
    .
s = malloc(20);
strcpy( s, "abc" );
```

Since **`malloc`** and **`strcpy`** are library routines, we would ordinarily obtain strong type violations when assigning the value returned by **`malloc`** to a strongly typed variable s or when passing the strongly typed s into strcpy. However, the l flag suppresses these strong type clashes.

Strong types can be used with bit fields. Bit fields of length one are assumed to be, for the purpose of strong type checking, the prevailing Boolean type if any. If there is no prevailing Boolean type or if the length is other than one, then, for the purpose of strong type checking, the type is the bulk type from which the fields are carved. Thus:

```
/*-strong(AJXb,Bool)     */
/*-strong(AJX,BitField)  */

typedef int Bool;
typedef unsigned BitField;

struct foo
    {
    unsigned a:1, b:2;
    BitField c:1, d:2, e:3;
    } x;
```

```
void f()
  {
    x.a = (Bool) 1;     /*   OK        */
    x.b = (Bool) 0;     /* strong type */
                        /*  violation  */
    x.a = 0;            /* strong type */
                        /*  violation  */
    x.b = 2;            /*   OK        */
    x.c = x.a;          /*   OK        */
    x.e = 1;            /* strong type */
                        /*  violation  */
    x.e = x.d;          /*   OK        */
    }
```

In the above, members **a** and **c** are strongly typed Bool, members **d** and **e** are typed **BitField** and member **b** is not strongly typed.

To suppress the Boolean assumption for one-bit fields use the flag 'f' in the **-strong** option for the Boolean. In the example above, this would be **-strong(AJXbf,Bool)**.

3.7.7.2 -index

-index *(flags,ixtype,sitype[, sitype]...*)

This option is supplementary to and can be used in conjunction with the **-strong** option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype* (or *sitype*s if more than one is provided). Both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a **typedef** declaration. *flags* can be

c	allow Constants as well as ixtype, to be used as indices.
d	allow array Dimensions to be specified without using an ixtype.

Examples of -index

For example:

```
/* -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) when used in array notation are also checked. 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:

```
/*-index(,Count,Temperature,Pressure,Volt-
age)*/

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 */

/* -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];
```

3.7.7.3 -parent

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*. Or 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.

The Natural Type Hierarchy

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

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

In this case **Flags1** can be combined freely with **Flags**, but not with **Flags2** or with **Flags3**. This depends, however, on the state of the fhs

(Hierarchy of Strong types) flag which is normally ON. If you turn it off with the

-fhs

option, the natural hierarchy is not formed.

We say that **FLAGS** is a *parent* type to each **of Flags1**, **Flags2** and **Flags3** which are its children. Being a parent to a child type is similar to being a base type to a derived type in an object oriented system with one very important difference. A parent is 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 is a subset of a derived type and assignment can go only one way.

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

```
/*-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;              /* Warning */
    if(f1 & f2)           /* Warning */
        f2 = f2 | F_ONE;  /*  OK     */
    f2 = F_ONE | f2;      /*  OK     */
    f2 = F_ONE | f1;      /* Warning */
    }
```

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 arbitrarily many 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 file 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;
   ...
```

Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via **typedef**'s has a limitation. All the types in a single tree must be the same underlying type. The -parent option can be used to supplement (or completely replace) the strong type hierarchy established via **typedef**s. An option of the form:

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

where *Parent* and *Child* are type names defined via **typedef** will create a link in the strong type hierarchy between the *Parent* and each of the *Child* types. The *Parent* is considered to be equivalent to each *Child* for the purpose of Strong type matching. The types need not be the same underlying type and normal checking between the types is unchanged.

A link that would form a loop in the tree will not be 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.

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 above example, you will capture the following hierarchy tree.

```
- Flags
  |
  |- Flags1
  |  |_ Small
  |
  |_ Flags2
```

The characters used to draw the hierarchy may be regular ASCII characters.

If the **-fhs** option is set (turning off the hierarchy of strong types flag) **typedef**'s will not add hierarchical links. The only links that will be formed will be via the **-parent** option.

3.7.7.4 Hints on Strong Typing

1. Beware of excessive casting. If, in order to pull off a system of strong typing you need to cast just about every access, you are missing the point. The casts will inhibit even ordinary checking which has considerable value. Remember, strong type-checking is gold, normal type-checking is silver, and casting is brass.
2. Rather than cast, use type hierarchies. For example:

   ```
   /*-strong(AXJ,Tight) -strong(,Loose) */
   typedef int Tight;
   typedef Tight Loose;
   ```

   Tight has a maximal amount of Strong Type checking; **Loose** has none. Since **Loose** is defined in terms of **Tight** the two types are interchangeable from the standpoint of Strong Type checking. Pre-

sumably you work with **Tight int**s most of the time. When absolutely necessary to achieve some effect **Loose** is used.

3. A time when it's really good to cast is to endow some otherwise neutral constant with a special type. **FZERO** of the previous section is an example.
4. For large, mature projects enter strong typing slowly working on one family of strong types at a time. A family of strong types is one hierarchy structure
5. Don't bother with making pointers to functions strong types. For example:

```
typedef int (*Func_Ptr)(void);
```

   If you make **Func_Ptr** strong, you're not likely to get much more checking that if you didn't make it strong. The problem is that you would then have to cast any existing function name when assigning to such a pointer. This represents a net loss of type-checking (remember: gold, silver, brass).
6. Rather than strong type a pointer, strong type the base type. For example:

```
typedef char TEXT;
typedef TEXT *STRING;
 TEXT buffer[100];
STRING s;
```

It may seem wise to strong type both **STRING** and **TEXT**. This would be a mistake since whenever you assign buffer to s, for example, you would have to cast. But note that **-strong(Ac, STRING)** would allow the assignment. It is usually better to strong type just **TEXT**. Then when buffer is assigned to **s** the indirect object **TEXT** is strongly checked and no cast is needed.

7. Care is needed in declaring strong self-referential **struct**s. The usual method, i.e.,

```
typedef struct list { struct list * next ;
... }
LIST;
```

   is incompatible with making **LIST** a strong type because its member **next** will not be pointer to strong. Better:

```
typedef struct list LIST;
struct list { LIST * next; ...};
```

   This is explicitly sanctioned in ANSI C and will make **next** compatible with other pointers to **LIST**.

3.7.7.5 Reference Information

Strong Expressions

An expression is strongly typed if:

1. it is a strongly typed variable, function, array, or member of **`union`** or **`struct`** or an indirectly referenced pointer to a strong type.
2. it is a cast to some strong type.
3. it is one of the type- propagating unary operators, (viz. + - ++ -- ~), applied to a strongly typed expression.
4. it is formed by one of the *balance* and *propagate* binary operators applied to two strongly typed expressions (having the same strong type). The balance and propagate operators consist of the five binary arithmetics (**`+ - * / %`**), the three bit-wise operators (**`& | ^^`**), and the conditional operator **`(? :`**).
5. it is a shift operator whose left side is a strong type.
6. it is a comma operator whose right side is a strong type.
7. it is an assignment operator whose left side is a strong type.
8. it is a *Boolean operator* and some type has been designated as Boolean (with a **`b`** or **`B`** flag in the **`-strong`** option). The Boolean operators consist of the four relationals (**`> >= < <=`**), the two equality operators **`(== !=`**), the two logical operators (**`|| &&`**), and unary **`!`**

General Information

When the option

`-strong` (*flags [, name]...*)

is processed, name and flags are entered into a so-called Strong Table created for this purpose.

If there is no name, then a variable, Default Flags, is set to the flags provided. When a subsequent **`typedef`** is encountered within the code, the Strong Table is consulted first and if the **`typedef`** name is not found, the Default Flags are used. These flags become the identifying flags for strong typing purposes for the type.

The option

`-index` (*flags, ixtype, sitype[,...]*)

is treated similarly. Each *sitype* is entered into the Strong Table (if not already there) and its index flags ORed with other strong flags in the

table. A pointer is established from *sitype* to *ixtype* which is another entry in the Strong Table.

For these reasons it does not, in general, matter in what order the **`-strong`** options are placed other than that they be placed before the associated **`typedef`**. There should be, at most, one option that specifies Default Flags.

Strong Types and Prototypes

If you are producing prototypes with some variation of the **`-od`** option (Output Declarations), and if you want to see the **`typedef`** types rather than the raw types, just make sure that the relevant **`typedef`** types are strong. You can make them all strong with a single option: **`-strong()`**. Since you have not specified '**A**', '**J**' or '**X**' you will not receive messages owing to strong type mismatches for Assigning, Joining or eXtraction. However, you may get them for declarations. You can set

```
-etd(strong)
```

to inhibit any such messages.

Epilogue

The hierarchy of strong types compels one to compare this hierarchy with the object-oriented hierarchy of C++ and other languages. If one closely examines the phase "code reusability" touted as a property of OOP one finds that it refers to the fact that one can write generic functions, i.e., functions that operate on more than just one **`struct`** type.

The way that one must do this in straight C is by passing pointers to a function expecting a **`void *`** pointer and hoping that all **`struc`**ts passed this way were compatible. With strict type checking (without hierarchies) you can not do it at all.

So perhaps it is now a little clearer why "code reusability" is such a puzzle to C programmers. With a loosely typed language such as C, code reusability was something you did not have to work very hard for. In the past, when C code would mix pointers and int's freely "code reusability" had always been a fact of life. Coming from Pascal or Ada, however, OOP really does provide for the writing of generic functions. One may speculate that C's general success over Pascal may be attributed in part to its greater code reusability. For C, OOP provides not reuse but type checking.

Viewed in this way the strong type hierarchies described in this paper serve the same purpose for scalars as the OOP hierarchies do for

structs. It may also be pointed out that for each **struct** one could have an associated **void** pointer and arrange these in a hierarchy. For example:

```
typedef void    *vShape;
typedef vShape  vCircle;
typedef vShape  vSquare;
```

With these as strong types a server routines could accept **vShape**s as arguments and provide for **vCircle**s and **vShape**s as results. The strong type facility would check types and keep them in line. All the benefits of object orientedness would result with one additional bonus. The server routines alone would know or care about the internal structure of a Shape, Circle or Square. This would greatly reduce header file cascading.

3.7.8 Other Options

When using *STATIC*, you can use the 'other' options. These options effect the behavior of *STATIC*. Some options are activated by toggles, while other must be keyed in.

3.7.8.1 Toggle Options

To select a toggle option:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu.
3. Drag the mouse to the **Other** submenu and select **Toggles**.
4. The **Other Toggles Options** window pops up.
5. To select an option, simply click on the corresponding check button. The default is **Unit Checkout -u**.
6. Click on **OK**.

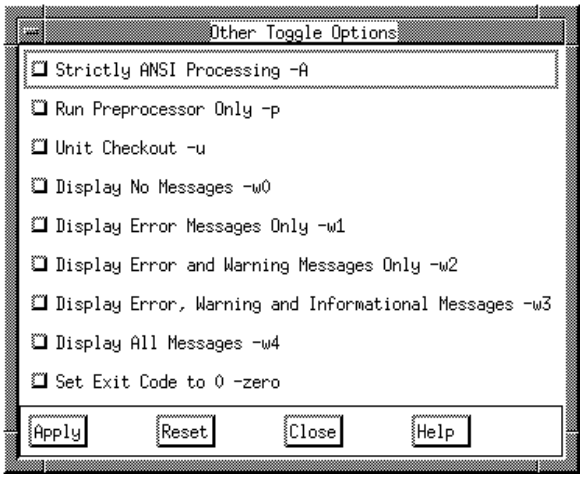

FIGURE 27 Other Toggle Options Window

The **Other Toggle Options** window consists of the following options:

- **Strictly ANSI Processing -A**: Enables Elective Note 950. Non-ANSI keywords (i.e., reserved words) and other non-ANSI features such as the / / form of comment are reported but duly processed according to their non-ANSI meaning. A common situation is when your compiler header files are non-ANSI but you want your ANSI program to be checked. For this situation, use, in addition to **-A**, the option **-elib(950)**. Note, to really check your code for ANSI compliance, use *STATIC* with a set of ANSI header files.
- **Unit Checkout -u**: This is defaulted on. It is used when running a subset (frequently just one) of the modules comprising a program. For example, **-u** suppresses the inter-module messages 526, 552, 628, 714, 729, 755-759, 765.
- **Run Preprocessor Only -p**: Runs just the preprocessor. If this flag is set, the entire character of *STATIC* is changed from a diagnostic tool to a preprocessor. Running *STATIC* with the **-os(file) -p** will produce on **file.p** the result of processing all the # lines within file.c.

The warning levels consist of:

- **Display No Messages -w0** – No messages (except for fatal errors)
- **Display Error Messages Only -w1** – No warnings or informationals. (Equivalent to **-e4?? -e5?? -e6?? -e7??**).
- **Display Error and Warning Messages Only -w2**: This is equivalent to **-e7??** and **-e8??**.
- **Display Error, Warning and Informational Messages Only -w3**
- **Display All Messages -w4** Equivalent to **+e9??**.

  Because options are processed in order, the combined effect of the two options: **-w2 +e720** is to turn off all Informational messages except 720.
- **Set Exit Code to 0 -zero** – This is useful to prohibit the premature termination of **make** files.

  With *STATIC*, you can also use define options.

3.7.8.2 Define Options

To select a define option:

1. Click on the **Options** pull-down menu.
2. Drag the mouse to the **Modify** submenu.
3. Drag the mouse to the **Other** submenu and select **Defines**.
4. The **Other Define Options** window pops up. When you add more options, you can use the scroll bars in the **Other Option Set** window to move up / down or side / side. The default option is -i / usr / include.

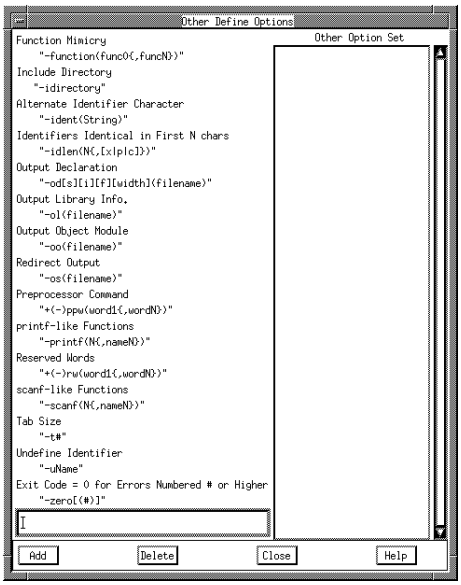

FIGURE 28 Other Define Options Window

5. The **Add** and **Delete** buttons allow you to add or delete compiler options to the **Other Option Set** list. To add an option:
   - Position the mouse pointer so it is in the specification region and click. A cursor should appear.
   - Type in the option you would like to add.
   - If you want the option listed at the bottom of the **Other Option Set**, click on **Add**.
   - If you want the option listed at a specific location in the **Other Option Set** window, highlight the option where you would like the new option to go below and then click on **Add**. The new option will be inserted below the option you highlighted.

To delete a switch:

- Highlight the switch you would like to remove.
- Click on **Delete**.
- The option should be removed.

These are the define options:

`-function`(*func0(, funcN)*This option specifies that *FuncN* are like *Function0* in that they exhibit special properties normally associated with

Func0. The special functions with built-in meaning are **abort**, **exit**, **free**, **longjmp**, **realloc**, and **setjmp**. See the section on function mimicry information (See Section 7.12 - "Function Mimicry (-function)" on page 183.).

-i*directory* Files not found in the current directory are searched for in the directory specified. There is no intrinsic limit to the number of such directories. The search order is given by the order of appearance of the **-i***directory* strings on the command line. 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**.

STATIC also supports the INCLUDE environment variable (See Section 9.2 - "include Processing" on page 192.). Note: Any directory specified by a **-i**directive takes precedence over the directories specified via the **INCLUDE** environment variable. *STATIC* also supports the **INCLUDE** environment variable for some systems where appropriate.

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

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

-ident*(*^@)

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

-idlen (*count[,options]*)

will report on pairs of identifiers in the same name space that are identical in their first count characters but are otherwise different. Options are:

x	linker (eXternal) symbols
p	Preprocessor symbols
c	Compiler symbols

If omitted, all symbols are assumed. Frequently, linkers and, less frequently, preprocessors and compilers, have a limit on the number of significant characters of an identifier. They will ignore all but the first n char-

acters. The **-idlen** option can be used to find pairs of identifiers that are identical in the first *n* characters but are nonetheless different. *STATIC* treats the identifiers as different but reports on the clash.

Option **p**, preprocessor symbols, refers to macros and parameters of function-like macros. Option **x**, linker symbols, refers to inter-module symbols. Option **c**, compiler symbols, refers to all the other symbols and includes symbols local to a function, **struct/union** tags and member names, enum constants, etc. Warning 621, **Identifier clash** may be suppressed for individual identifiers with the **-esym** option. **-idlen** is OFF by default.

-od[s][i][f][*width*](*filename*)

Output Declarations (including prototypes) to *filename* using the optional *width* to specify the maximum line width. If **i** is specified, functions with internal linkage are included; if **s** is specified, structure definitions are provided and, if **f** is specified, output is restricted to functions. **[s][i][f]** may appear in any order. (See Section 7.6 - "Prototype Generation" on page 172.)

-oo[(*filename)*] Output Object Module to filename. This option causes binary information for all processed modules (usually just one) to be output to filename. The extension for filename should be **.lob**. If filename is omitted, as in **-oo**, a name will be manufactured using the first name of the source file and an extension of **.lob**. (See Section 6.3 - "Producing a LOB" on page 164.) Related options are **+fol** and **+fod**.

-os(*filename*) Causes Output directed to Standard out to be placed in the file *filename*. This is like redirection and has the following advantages: (a) not all systems support redirection and (b) redirection can have strange side effects (see Section 6.4 for make file information).

+ppw(*word1[,wordN]...*)

Adds

-ppw(*word1[,wordN]...*)

Removes preprocessor command word(s) *word1*, *wordN*, etc. *STATIC* might stumble over strange preprocessor commands that your compiler happens to

support (for example some UNIX system compilers support **#ident**). Since this is something that can-NOT be handled by a suitable **#define** of some identifier we have added the **+ppw** option. For example, **+ppw(ident)** will add the preprocessor command alluded to above. *STATIC* then recognizes and ignores lines beginning with **#ident**.

-printf(*N{ ,nameN}*)

This option specifies that *name1*, *nameN*, etc. are functions which take `printf`-like formats. The format is provided in the Nth argument. For example, *STATIC* 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 following the Nth argument are expected to agree in size and type specified by the format. See also **-scanf** below and a later section (See Section 7.12 - "Function Mimicry (-function)" on page 183.) for function mimicry information.

+rw(*word1[,wordN]...*)

Adds

-rw(*word1[,wordN]...*)

Removes Reserved Word(s) *word1*, *wordN*, etc. If the meaning of a reserved word being added is already known, that meaning is assumed. For example, **+rw(fortran)** will enable the reserved word **fortran**. If the reserved word has no prior known semantics, then it will be passed over when encountered in the source text. For example:

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

adds the three reserved words shown. **_loadds** is assigned a meaning consistent with that of the Microsoft C compiler (See Section 3.7.5 - "Compiler Vendor Options" on page 54.). asm is assigned a meaning consistent with that of the Turbo C compiler. **entry** is assigned no meaning; it is simply skipped over when encountered in a source statement. Since no meaning is to be ascribed to entry, it

could just as well have been assigned a null value as in

-dentry=

As a special case, if *wordn* is ***ms**, then all the Microsoft keywords are identified. Thus **+rw(*ms)** adds all the Microsoft keywords. (See Section 8.3 - "Additional Reserved Words" on page 190.) This would not normally be necessary for Microsoft users since co-msc.lnt has the -cmsc option embedded within it and this option also enables the Microsoft keywords. However, users of other compilers may wish to enable these keywords because they have become something of a de-facto standard.

By default, a number of Microsoft's keywords are pre-enabled because they are so commonly used. To deactivate all of them use **-rw(*ms)**. See the section that describes **-cmsc** (See Section 3.7.5 - "Compiler Vendor Options" on page 54.) for the current list of supported Microsoft keywords (reserved words).

-scanf(*N{, nameN}*)

This option specifies that *nameN* is a function which takes **scanf**-like formats. The format is provided in the Nth argument. For example, *STATIC* is preconfigured as if the following options were given:

-scanf(1,scanf)

-scanf(2,sscanf,fscanf)

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

-t# Sets *STATIC*'s idea of what the tab size is. This is used for indentation checking. By default *STATIC* presumes that tabs occur every 8 column positions. If your editor is converting blanks to tabs at some other exchange rate, then use this option. For example **-t4** indicates that a tab is worth 4 blank characters.

-u *Name* Can be used to undefine an identifier that is normally pre-defined. For example:

-u_lint

will undefine the identifier **_lint** which is normally pre-defined before each module. The undefine will take place for all subsequent modules after the default pre-definitions are established. If given within a comment, the undefine will take place immediately as well as in subsequent modules (similar to **-d..**). The observant reader will notice that you may not undefine the name **nreachable**.

-zero[(*#*) Will set the exit code to zero if all reported errors are numbered # or higher. More precisely, errors which have an error message number that 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.

3.8 Saving Modifications

After you modify the different categories of available options, you can decide to save these modifications to a new file or overwrite one the category files specified in the configuration file *static.rc*. *STATIC* automatically reads in the default configuration file *static.rc*.

If your want your changes to be permanent, it is best to simply save the modifications to one the configuration's file's existing category of options.

If you modify *STATIC*'s error messages, for instance, and want these options to be permanent, you can simply override these changes to the existing *sr.err* file. The configuration file *static.rc* lists *sr.err* as the file where the default error message options are listed. Now, the next time you invoke *STATIC* and select a file for analysis, *static.rc* will automatically be loaded with the new option modifications. As a precautionary measure, you should make a copy of the original sr.err file (i.e., cp sr.err sr.err.old.)

If you want your modifications to apply only to certain situations, you probably don't want it saved to one of the configuration file categories. Instead, you should save these kind of modifications to a new file name. When *STATIC* is invoked the existing *static.rc* is left intact.

To activate these modifications that you save to a new file, you must use the Load utility (see the following "Leading Option Modification Files" Section) before you select a file for analysis. To save option modifications, you must exit out of *STATIC*. (See Section 3.10 - "Exiting STATIC" on page 91.)

Loading Option Modification Files

Modifications can either be permanent (if saved to a file within the configuration file) or used for certain situations (usually saved to a new file). When you save your option modifications to a new file and re-invoke *STATIC*, you will notice that the old options are left intact. This is because *STATIC* is reading in the *static.rc* configuration file. To activate your new modifications:

1. Click on the **Options** pull-down menu.
2. Select **Load**.
3. The **Load** window pops up. It lists the different categories of options.
4. If you made and saved modifications to the error messages and want the new list activated, you would click on the **Error** radio button.

There are corresponding radio button to flag options, library header file options, size options, strong types, and other options.

FIGURE 29 Load Window

5. Click on **OK**.

   The Load window disappears and a **Load Error Option** file selection dialog box pops up. There are corresponding dialog boxes for the other option categories.
6. This window is similar to most file selection dialog boxes, except it has an **Options Set** window. When a file is selected, it will list the options in that file.
7. Select the file where you saved the changes to.
8. The options for that file should now be listed in the **Options Set** window. *STATIC* will now read the options from the file specified instead of *static.rc*.
9. If you changed other categories of options and want these options activated, then follow the steps 1 - 8.
10. You can new select a source code file for analysis and *STATIC*'s report should reflect the new file's options.

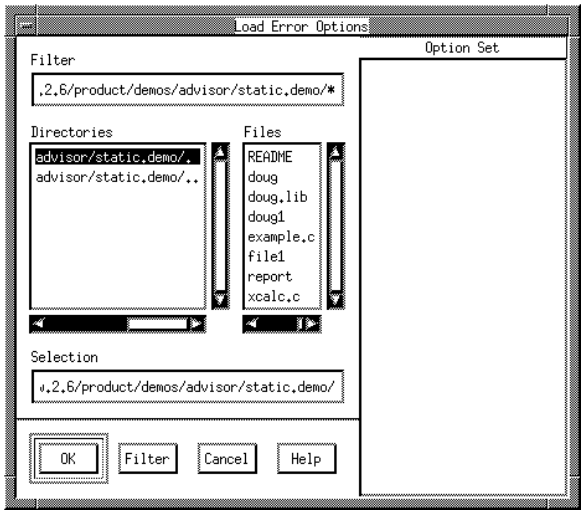

FIGURE 30 Load Error Options Window

3.9 Customizing STATIC

Modifying options through the GUI is most beneficial when you want to make minor modifications to the default option settings. When you make major modifications, its usually easiest just to edit the configuration file *static.rc*.

static.rc looks like this:

```
CompilerOptionFile = static.cmp
ErrorOptionFile = static.err
FlagOptionFile = static.flg
FormatOptionFile = static.frm
LibraryOptionFile = staticb
OtherOptionFile = static.oth
SizeOptionFile = static.size
StrongOptionFile = static.stg
```

It basically consists of several default files, which represent the different categories of options (similar to the GUI). represent the different categories of options (similar to the GUI).

If you want to remove categories of options, you simply edit the *static.rc* file accordingly.

If you want to edit one of the category's options, edit the corresponding file. If you wanted to suppress or reactivate error messages, for instance, you would edit static.err (shown below) using any UNIX text editor (such as vi).

```
-e746
-e534
-e762
-e578
-elib(537)
-elib(544)
-elib(762)
-elib(652)
-elib(760)
-esym(516,XtAddCallback)
-esym(718,fprintf)
-esym(515,fprintf)
-esym(718,unlink,printf)
-esym(715,w,callData)
-esym(718,system)
-esym(515,sprintf)
-esym(515,printf)
```

```
-esym(516,printf)
-esym(718,fputs,fputc)
-esym(558,fprintf)
-esym(718,fread)
-esym(718,fclose)
-esym(526,fprintf)
-esym(526,fread)
-esym(526,fclose)
```

3.10 Exiting STATIC

The **Exit** option allows you to close the **Main** window as well as save option modifications. Here's how:

1. Click on the **File** pull-down menu.
2. Select **Exit**.
3. If you made a modification to any of the option categories, you will be prompted with a dialog box telling which category was changed and if you want to save those changes. A different dialog box will pop for each category you modify. If you did not make modifications, please go to #7.
4. Click on **OK** if you want to save your changes; click on No if you don't want to save your changes.
5. If you clicked on **OK**, a file selection dialog box pops up.
6. For permanent changes, overwrite *static.rc*'s corresponding file. If you made changes to error options, you would select *static.err*. If you modified a category for a particular circumstance and don't want to overwrite *static.rc*'s files, simply name a new file name. (See Section 3.8 - "Saving Modifications" on page 87.)

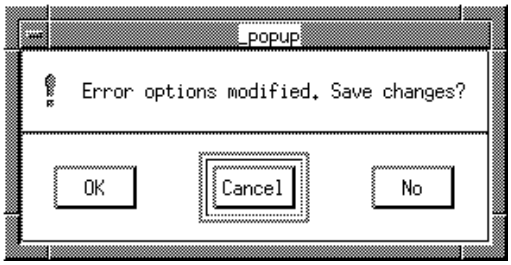

FIGURE 31 Saving Option Modifications

7. After modifications are saved or discarded, *STATIC* will prompt you to exit with a dialog box.

8. Click on **OK**.

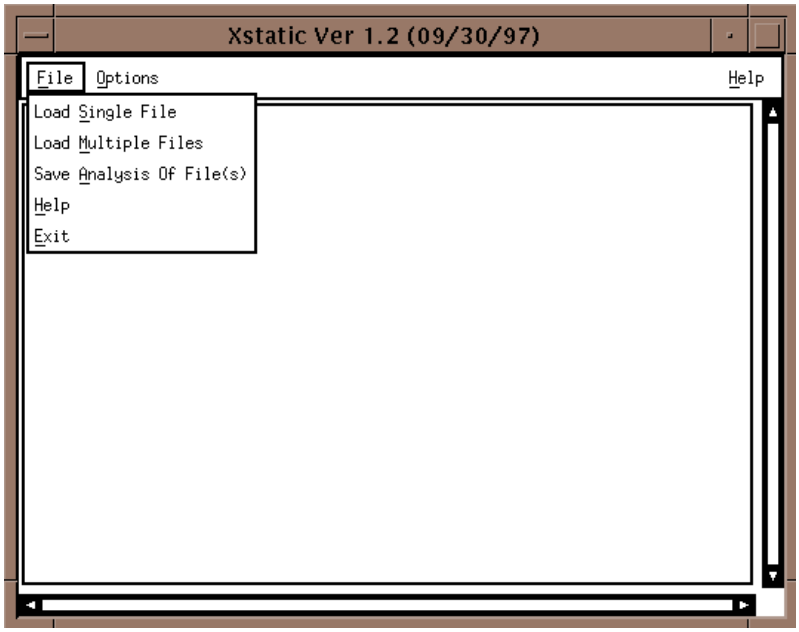

FIGURE 32 Exiting STATIC

CHAPTER 4

Messages

This chapter details all of the error messages *STATIC* produces. When *STATIC* produces a report for your source code file, use this chapter to reference the error message number.

4.1 Categories of Messages

Most messages have an associated number. By looking up the number in the list below you can obtain additional information about the cause of the message.

Here are the categories of messages:

- Errors in the 1-199 range are syntax errors.
- Errors in the 200-299 range are *STATIC* internal error and should never occur.
- Errors in the 300-399 range are fatal errors usually brought about by exceeding some limit.
- Messages in the 400-699 range are warning messages. They indicate that something is likely to be wrong with the C program being examined.
- Messages in the 700-799 range are informational messages. These may be errors but they also may represent legitimate programming practices depending upon personal programming style.
- Messages in the 900-999 range are called Elective Notes. They are not automatically generated. You may examine the list to see if you wish to be alerted to any of them. To turn on any of these messages, please refer to the correct section (See Section 3.7.1 - “Error Messages Options” on page 32.).

4.2 Message Glossary

The terms used to describe the messages are:

argument — The actual argument of a function as opposed to a dummy (or formal) parameter of a function (see **parameter**).

arithmetic — Any of the integral types (see below) plus **float**, **double**, and **long double**.

Boolean — In general, 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 operan** where **op** is a relational (**> >= < <=**), an equality operator (**== !=**), logical **And (&&)** or logical **Or** (| |). A context is said to require a Boolean if it is used in an **if** or **while** clause or if it is the 2nd expression of a **for** clause or if it is an argument to one of the operators **: &&** or **||**. An expression needn't be definitely Boolean to be acceptable in a context that requires a Boolean. Any integer or pointer is acceptable.

declaration — Gives properties about an object or function (as opposed to a definition).

definition — That which allocates space for an object or function (as opposed to a declaration) and which may also indicate properties about the object. There should be only one definition for an object but there may be many declarations.

integral — A type that has properties similar to integers. These include char, short, int, and long and unsigned variations of any of these.

scalar — Any of the arithmetic types plus pointers.

1value — 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 — Subelements of **structs** and **unions** 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`** file plus any text within any **`#include`** file(s).

parameter A formal parameter of a function as opposed to an actual argument (see **argument**). Some of the messages are parameterized with one or more of the following italicized names:

Char Some character

Context Specifies one of several contexts in which an assignment can be made. Can be one of:

- assignment--refers to an explicit assignment operator.
- return--refers to the implied assignment of a return statement. The type of the expression is converted implicitly to the type of the function.
- initialization--refers to the assignment implied by an initialization statement.
- arg. no...--refers to the implied assignment of an argument in the presence of a prototype. The type of the expression is implicitly converted to the type within a prototype.

FileName A filename. Messages containing this parameter can be suppressed with the **`-efile`** ...) option.

Int Some integer.

Location A line number followed optionally by a filename (if different from the current) and/or a module name if different from the current.

String A sequence of characters identified further in the message description.

Symbol The name of a user identifier referring to a C object such as variable, function, structure, etc. Messages containing this parameter can be suppressed with the -esym(...) option.

Type A type or a top type base is provided. A top type base is one of **`pointer`**, **`function`**, **`array`**, `struct`, **`union`**, or `enum`.

TypeDiff Specifies the way in which one type differs from another. Because of type qualification, function prototypes, and type compounding, it may not be obvious how two types differ. Also, see the **`-etd`** option to in-

hibit errors based on type differences. **TypeDiff** can be one or more of:

- **basic**--The two types differ in some fundamental way such as double versus int.
- **count**--Two function types differ in the number of arguments.
- **ellipsis**--Two function types differ in that one is prototyped using an ellipsis and the other is not prototyped.
- **incomplete**--At least one of the types is only partially specified such as an array without a dimension or a function without a prototype.
- **nominal**--The types are nominally different but are otherwise the same. For example, int versus long where these are the same size or double versus long double where these are the same size. The two types are either both integral or both float or are functions that return types or have arguments that differ nominally. If long's are the same size as int's then unsigned long will differ from int both as nominal and as signed / unsigned. If not the same size, then the difference is precision.
- **origin**--The types are not actually different but have different origins. For example a **struct** is defined in two separate modules rather than in one header file. If for some reason you want to do this then use the option `-etd(origin)`.
- **precision**--Two arithmetic types differ in their precision such as int vs. long where these are different sizes.
- **promotion**--Two function types differ in that one is prototyped with a char, short or float type and the other is not prototyped.
- **ptrs to...**--Pointers point to different types, some TypeDiff code follows.
- **ptrs to incompatible types**--Pointers point to types which in turn differ in precision, count, size, ellipsis or promotion.
- **qualification**==Qualifiers such as **const**, **volatile**, etc. are inconsistent.
- **signed/unsigned**--The types differ in that one is a signed integral type and the other is unsigned of the same size, or they are both functions that return types that differ in this way, or they are both pointers to types that differ in this way.
- **size**--Two arrays differ in array dimension.

- **`strong`**--Two types differ in that one is strong and the other is not the same strong type.
- **`void/nonvoid`**--The two types differ in that one is void and the other is not or, more frequently, they are both functions returning types that differ in this respect or pointers to types that differ in this respect.
- **`int/enum`**--One type is an **`enum`** and the other is an **`int`**.
- **`Type = Type`**--The two types in an assignment of some kind differ in some basic way and no more information is available.
- **`Type vs. Type`**--The two types differ in some basic way and no more information is available.

4.3 Syntax Error Messages

1 **Unclosed Comment** (*Location*)--End of file was reached with an open comment still unclosed. The Location of the open comment is shown.

2 **Unclosed Quote**--An end of line was reached and a matching **quote character** (*single or double*) to an earlier quote character on the same line was not found.

3 **#else without a #if** --A **#else** was encountered not in the scope of a **#if**, **#ifdef** or **#ifndef**.

4 **Too many #if levels**--An internal limit was reached on the level of nesting of **#if**'s (including **#ifdef**'s and **#ifndef**'s).

5 **Too many #endif's--A #endif** was encountered not in the scope of a #if or #ifdef or #ifndef.

6 **Stack Overflow** --One of the built-in non-extendible stacks has been overextended. The possibilities are too many nested **#if**s, **#include**s (including all recursive **#include**s), static blocks (bounded by braces) or **#define** replacements.

7 **Unable to open include file:***FileName FileName* is the name of the include file which could not be opened. See also **flag fdi** (See Section 3.7.2 - “Flag Options” on page 38.) and option -i... (See Section 3.7.6 - “Compiler Customization Options” on page 58.).

8 **Unclosed #if** (*Location*) A **#if** (or **#ifdef** or **#ifndef**) was encountered without a corresponding **#endif**. Location is the location of the **#if**.

9 **Too many #else**'s in **#if** (*Location*) A given**#if** contained a **#else** which in turn was followed by either another **#else** or a **#elif**. The error message gives the line of the **#if** statement that started the conditional that contained the aberration.

10 **Expecting String** *String* is the expected token. The expected token could not be found. This is commonly given when certain reserved words are not recognized.

```
int __interrupt f();
```

will receive an **Expecting** ';' message at the **f** because it thinks you just declared __interrupt. The cure is to establish a new reserved word +rw(__interrupt). Also, make sure you are using the correct compiler options file.

11 **Excessive Size** The filename specified on a **#include** line had a length that exceeded **FILENAME_MAX** characters.

12 **Need < or After a #include** is detected and after macro substitution is performed, a file specification of the form <filename> or filename is expected.

13 **Bad type** A type adjective such as long, unsigned etc. cannot be applied to the type which follows.

14 **Symbol previously defined** (*Location*) The named object has been defined a second time. The location of the previous definition is provided. If this is a tentative definition (no initializer) then the message can be suppressed with the +fmd flag. (See Section 3.7.2 - "Flag Options" on page 38.).

15 *Symbol* **redeclared** *(TypeDiff) (Location)* The named symbol has been previously declared or defined in some other module (location given) with a type different from the type given by the declaration at the current location. The parameter *TypeDiff* provides further information on how the types differ (see glossary above).

16 **Unrecognized name** A# directive is not followed by a recognizable word. If this is not an error, use the **+ppw** option (See Section 3.7 - "Modifying the Report Options" on page 32.).

17 **Unrecognized name** A non-parameter is being declared where only parameters should be.

18 *Symbol* **redeclared** (*TypeDiff*) **conflicts with** *Location* A symbol is being redeclared. The parameter *TypeDiff* provides further information on how the types differ (see Glossary above). *Location* is the location of the previous definition.

19 **Useless Declaration** A type appeared by itself without an associated variable, and the type was not a `struct` and not a union and not an `enum`.

20 **Illegal use of = , ignored** A function declaration was followed by an = sign.

21 **Expected {** An initializer for an indefinite size array must begin with a left brace.

22 **Illegal operator** A unary operator was found following an operand and the operator is not a post operator.

23 **Expected colon** A **? operator** was encountered but this was not followed by a : as was expected.

24 **Expected an expression** An operator was found at the start of an expression but it was not a unary operator.

25 **Illegal constant** Too many characters were encountered in a character constant (a constant bounded by' marks).

26 **Expected an expression** An expression was not found where one was expected.

27 **Illegal character** (*Oxhh*) An illegal character was found in the source code. The hex code is provided in the message. A blank is assumed.

28 **Redefinition of symbol***Symbol Location* The identifier preceding a colon was previously declared at the Location given as not being a label.

30 **Expected a constant** A constant was expected but not obtained. This could be following a case keyword, an array dimension, bit field length, enumeration value, #if expression, etc.

31 **Redefinition of symbol***Symbol* conflicts with Location A data object or function previously defined in this module is being redefined.

32 **Bad field size** The length of a field was given as non-positive, (0 or negative).

33 **Illegal constant** A constant was badly formed as when an octal constant contains one of the digits 8 or 9.

34 **Non-constant initializer** A non-constant initializer was found for a static data item.

35 **Initializer has side-effects** An initializer with side effects was found for a static data item.

36 **Redefining the storage class of symbol** *Symbol* **conflicts with** *Location* An object's storage class is being changed.

38 **Redefinition of symbol** *Symbol* An element of a structure or unionis being redefined.

39 **Redefinition of symbol** *Symbol* **conflicts with** *Location* A `struct` or union is being redefined.

40 **Undeclared identifier** (*String*) Within an expression, an identifier was encountered that had not previously been declared and was not followed by a left parenthesis. String is the name of the identifier.

41 **Redefinition of symbol** *Symbol* A parameter of either a function or a macro is being repeated.

42 **Expected a statement** A statement was expected but a token was encountered that could not possibly begin a statement.

43 **Vacuous type for variable** *Symbol* A vacuous type was found such as an array with no bounds or a structure with no members in a context that expected substance.

44 **Need a switch** A case or default statement occurred outside a switch.

45 **Bad use of register** A variable is declared as a register but its type is inconsistent with it being a register such as a function.

46 **Field type should be int** Bit fields in a structure should be typed unsigned or int. If your compiler allows other kinds of objects, such as char, then simply suppress this message.

47 **Bad type** Unary minus requires an arithmetic operand.

48 **Bad type** Unary * or the left hand side of the ptr (->) operator requires a pointer operand

49 **Expected a type** Only types are allowed within prototypes. A prototype is a function declaration with a sequence of types within parentheses. The processor is at a state where it has detected at least one type within parentheses and so is expecting more types or a closing right parenthesis.

50 **Expected an lvalue** Unary & operator requires an value (a value suitable for placement on the left hand side of an assignment operator).

51 **Expected integral type** Unary ~ expects an integral type (**signed** or **unsigned char**, **short**, **int**, or **long**).

52 **Expected an lvalue** autodecrement (--) and autoincrement (++) operators require an lvalue (a value suitable for placement on the left hand side of an assignment operator). Remember that casts do not normally produce lvalues. Thus

```
++(char *)p;
```

is illegal according to the ANSI standard. This construct is allowed by some compilers and is allowed if you use the **+fpc** option (Pointer Casts are lvalues). See the correct section for flag options (See Section 3.7.2 - “Flag Options” on page 38.)

53 **Expected a scalar** Autodecrement (--) and autoincrement(++) operators may only be applied to scalars (arithmetics and pointers).

54 **Division by 0** The constant 60 was used on the right hand side of the division operator (/) or the remainder operator (%).

55 **Bad type** The context requires a scalar, function, array, or `struct` (unless -fsa).

56 **Bad type** Add/subtract operator requires scalar types and pointers may not be added to pointers.

57 **Bad type** Bit operators (&, | and ^^) require integral arguments.

58 **Bad type** Bad arguments were given to a relational operator; these always require two scalars and pointers can't be compared with integers (unless constant 0).

59 **Bad type** The amount by which an item can be shifted must be integral.

60 **Bad type** The value to be shifted must be integral.

61 **Bad type** The context requires a Boolean. Booleans must be some form of arithmetic or pointer.

62 **Incompatible type** (*TypeDiff*) for operator: The 2nd and 3rd arguments to ? : must be compatible types.

63 **Expected an lvalue** Assignment expects its first operand to be an lvalue.

64 **Type mismatch** *(Context) (TypeDiff)* There was a mismatch in types across an assignment (or implied assignment, see Context).*TypeDiff* specifies the type difference. See options **-epn, -eps, -epu, -epp** See the correct section for error inhibition options (See Section 3.7.1 - "Error Messages Options" on page 32.). to suppress this message when assigning some kinds of pointers.

65 **Expected a member name** After a dot (.) or pointer (->) operator a member name should appear.

66 **Bad type** A void type was employed where it is not permitted. If a void type is placed in a prototype then it must be the only type within a prototype. (See error number 49).

67 **Can't cast from** *Type* **to** *Type* Attempt to cast a non-scalar to an integral.

68 **Can't cast from** *Type* **to** *Type* Attempt to cast a non-arithmetic to a float.

69 **Can't cast from** *Type* **to** *Type* Bad conversion involving incompatible structures or a structure and some other object.

70 **Can't cast from** *Type* **to** *Type* Attempt to cast to a pointer from an unusual type (non-integral).

71 **Can't cast from** *Type* **to** *Type* Attempt to cast to a type that does not allow conversions.

72 **Bad option** '*String*' Was not able to interpret an option. The option is given in *String*.

73 **Bad left operand** The cursor is positioned at or just beyond either an -> or a . operator. These operators expect an expression primary on their left. Please enclose any complex expression in this position within parentheses.

74 **Address of Register** An attempt was made to apply the address (&) operator to a variable whose storage class was given as register.

75 **`Too late to change sizes`**(**`option`**'*String*') The size option was given after all or part of a module was processed. Make sure that any option to reset sizes of objects be done at the beginning of the first module processed or on the command line before any module is processed.

76 **`Can't open file`**: *String String* is the name of the file. The named file could not be opened for output. The file was destined to become a *STATIC* object module.

78 **`Symbol`** *Symbol* **`typedef'ed at`** *Location* used in expression The named symbol was defined in a typede statement and is therefore considered a type. It was subsequently found in a context where an expression was expected.

79 **`Bad type for % operator`** The % operator should be used with some form of integer.

80 **`this use of ellipsis is not strictly ANSI`** The ellipsis should be used in a prototype only after a sequence of types not after a sequence of identifiers. Some compilers support this extension. If you want to use this feature suppress this message.

81 `struct` / union not permitted in equality comparison Two `struct`'s orunion's are being compared with one of ==or !=. This is not permitted by the ANSI standard. If your compiler supports this, suppress this message.

82 **`return <exp>; illegal with void function`** The ANSI standard does not allow an expression form of the return statement with a void function. If you are trying to cast to void as in return (void)f(); and your compiler allows it, suppress this message.

83 **`Incompatible pointer types with subtraction`** Two pointers being subtracted have indirect types which differ. You can get *STATIC* to ignore slight differences in the pointers by employing one or more of the -ep options described in the section that details error inhibition options (See Section 3.7.2 - “Flag Options” on page 38.).

101 **`Expected an identifier`** While processing a function declarator, a parameter specifier was en-

countered that was not an identifier, whereas a prior parameter was specified as an identifier. This is mixing old-style function declarations with the new-style and is not permitted. For example

```
void f(n,int m)
```

will elicit this message.

102 **Illegal parameter specifications** Within a function declarator, a parameter must be specified as either an identifier or as a type followed by a declarator.

103 **Unexpected declaration** After a prototype, only a comma, semi-colon, right parenthesis or a left brace may occur. This error could occur if you have omitted a terminating character after a declaration or if you are mixing old-style parameter declarations with new-style prototypes.

104 **Conflicting types** Two consecutive conflicting types were found such as int followed by double. Remove one of the types!

105 **Conflicting modifiers** Two consecutive conflicting modifiers were found such as far followed by near. Remove one of the modifiers!

106 **Illegal constant** A string constant was found within a preprocessor expression as in

```
#if ABC == abc
```

Such expressions should be integral expressions.

107 **Label Symbol** (*Location*) not defined The Symbol at the given Location appeared in a goto but there was no corresponding label.

108 **Invalid context** A **continue** or **break** statement was encountered without an appropriate surrounding context such as a for, while, or do loop or, for the break statement only, a surrounding switch statement.

110 **Attempt to assign to void** An attempt was made to assign a value to an object designated (possibly through a pointer) as void.

111 **Assignment to const object** An object declared as const was assigned a value. This could arise

via indirection. For example, if p is a pointer to a const int then assigning to *p will raise this error.

113 **Inconsistent enum declaration** The sequence of members within an `enum` (or their values) is inconsistent with that of another `enum` (usually in some other module) having the same name.

114 **Inconsistent structure declaration for tag** *Symbol* The sequence of members within a structure (or union) is inconsistent with another structure (usually in some other module) having the same name.

115 **Struct/union not defined** A reference to a structure or a union was made that required a definition and there is no definition in scope. For example, a reference to p->a where p is a pointer to a `struct` that had not yet been defined in the current module.

116 **Inappropriate storage class** A storage class other than **`register`** was given in a section of code that is dedicated to declaring parameters. The section is that part of a function, preceding the first left brace.

117 **Inappropriate storage class** A storage class was provided outside any function that indicated either auto or register. Such storage classes are appropriate only within functions.

118 **Too few arguments for prototype** The number of arguments provided for a function was less than the number indicated by a prototype in scope.

119 **Too many arguments for prototype** The number of arguments provided for a function was greater than the number indicated by a prototype in scope.

122 **Illegal octal digit**(*Char*) The indicated character was found in a constant beginning with zero. Such constants are octal constants and should contain only octal digits.

123 **Macro** *(Symbol)* **defined with arguments at** *Location* The name of a macro defined with arguments was subsequently used without a following. The use of a macro should be consistent with its definition. It is not uncommon to suppress this message (with -e123), because some compilers allow, for example, the macro max() to coexist with a variable max.

124 Pointer to void not allowed A pointer to void was used in a context that does not permit void. This includes subtraction, addition and the relationals (> >= < <=).

125 **Too many storage class specifiers** More than one storage class specifier (static, extern, typedef, register or auto) was found. Only one is permitted.

126 **Inconsistent structure definition** (*Symbol*) The named structure (or union or enum) was inconsistently defined across modules. The inconsistency was recognized while processing a lint object module. Line number information was not available with this message. Alter the structures so that the member information is consistent.

127 **Illegal constant** An empty character constant ('') was found.

128 **Pointer to function not allowed** A pointer to a function was found in an arithmetic context such as subtraction, addition, or one of the relationals (> >= < <=).

129 **declaration expected, identifier** *Symbol* **ignored** In a context in which a declaration was expected an identifier was found. Moreover, the identifier was not followed by '(' or a '['

130 **Expected integral type** The expression in a **switch** statement must be some variation of an **int** (possibly **long** or **unsigned**) or an enum.

131 **syntax error in call of macro** *Symbol* **at location** *Location* This message is issued when a macro with arguments (function-like macro) is invoked and an incorrect number of arguments is provided. Location is the location of the start of the macro call. This can be useful because an errant macro call can extend over many lines.

132 **Expected function definition** A function declaration with identifiers between parentheses is the start of an old-style function definition (K&R style). This is normally followed by optional declarations and a left brace to signal the start of the function

body. Either replace the identifier(s) with type(s) or complete the function with a function body.

133 **`Too many initializers for aggregate`** In a brace-enclosed initializer, there are more items than there are elements of the aggregate.

134 **`Missing initializer`** An initializer was expected but only a comma was present.

135 **`Expected function definition comma`** assumed in initializer A comma was missing between two initializers. For example:

```
int a[2][2] = { { 1, 2 } { 3, 4 } };
```

is missing a comma after the first right brace (}).

136 **`Illegal macro name`** The ANSI standard restricts the use of certain names as macros. defined is on the restricted list.

137 **`constant`** *String* **`used twice within switch`** The indicated constant was used twice as a case within a switch statement. Currently only enumerated types are checked for repeated occurrence.

4.4 Internal Errors

200-299	Some inconsistency or contradiction was discovered in the *STATIC* system. This may or may not be the result of a user error. This inconsistency should be brought to the attention of Software Research.

4.5 Fatal Errors

Errors in this category are normally fatal and suppressing the error is normally impossible. However, those errors marked with an asterisk(*) can be suppressed and processing will be continued. For example **`-e306`** will allow reprocessing of modules.

302 **`Exceeded Available Memory`** Main memory has been exhausted. Try preprocessing separately.

303 **`String too long`** A single **`#define`** definition or macro invocation exceeded an internal limit (of 2048 characters).

304 **`Corrupt object file`** A *STATIC* object file is apparently corrupted. An expected header was not found. Please delete the object module and recreate it using the **`-oo`** option. See the section that describes producing a LOB (See Section 6.3 - “Producing a LOB” on page 164.).

305 **`Unable to open module:`** *FileName--FileName* is the name of the file. The named module could not be opened for reading. Perhaps you misspelled the name.

* 306 **`Previously encountered module:`** *FileName FileName* is the name of the module. The named module was previously encountered. This probably is not a user blunder.

307 **`Can't open indirect file:`***FileName--FileName* is the name of the indirect file. The named indirect file (ending in **`.lnt`**) could not be opened for reading.

308 **`Can't write to standard out--stdout`** was found to equal NULL. This is most unusual.

* 309 **`#error`** ... The **`#error`** directive was encountered. The ellipsis reflects the original line. Normally processing is terminated at this point. If you set the fce (continue on #error) flag, processing will continue.

310 **`Declaration too long`**: ‘*String...*' A single declaration was found to be too long for an internal buffer (about 2000 characters). The first 30 characters of the declaration is given in String. Typically this is caused by a very long `struct` whose sub strucs, if any, are untagged. First identify the declaration that is causing the difficulty. If a `struct` or union, assign a tag

to any unnamed substructs or sub unions. Typedef's can also be used to reduce the size of such declarations.

311 '*String*' **was one word too many** The number of reserved words exceeded an internal limit. This was brought about by too many +rw() or +ppw() options. (See Section 3.7.7 - "Strong Typing Options" on page 61.)

312 **Static Object Module has obsolete or foreign version id** A *STATIC* object module was produced with a prior or different version of *STATIC*. Delete the **.lob** file and recreate it using your new version of *STATIC*.

313 **Too many files** The number of files that *STATIC* can process has exceeded an internal limit. To process more files you will need to acquire a special version of *STATIC*. Please make inquiries to Software Research. The number of files is limited to 2048.

4.6 Warning Messages

401 **redefining the storage class of symbol** *Symbol* **conflicts with** *Location* The indicated *Symbol* declared static was previously declared without the static storage class. This is technically a violation of the ANSI standard. Some compilers will accept this situation without complaint and regard the *Symbol* as static.

402 **static function** *Symbol (Location)* not defined The named Symbol was declared as a static function in the current module and was referenced but was not defined (in the module).

403 **static symbol** *Symbol* **has unusual type modifier** Some type modifiers such as **_export** are inconsistent with the static storage class.

404 `struct` **not completed within file** *FileName* A `struct`(or **union** or `enum`) definition was started within a header file but was not completed within the same header file.

405 **#if not closed off within file** *FileName* An #if construct was begun within a header file (name given) but was not completed within that header file. Was this intentional?

406 **Comment not closed off within file** F*ileName* A comment was begun within a header file (name given) but was not completed within that header file. Was this intentional?

407 **Inconsistent use of tag** *Symbol* **conflicts with** *Location* A tag specified as a union, `struct` or `enum` was respecified as being one of the other two in the same module.

For example:

```
struct tag *p;
union tag *q;
```

will elicit this message.

408 **Type mismatch with switch expression** The expression within a case does not agree exactly with the type within the switch expression. For example, an enumerated type is matched against an **int**.

409 **Expecting a pointer or array** An expression of the form i [...] was encountered where i is an integral expression. This could be legitimate depending on the subscript operand. For example, if i is an int and a is an array then **i[a]** is legitimate but unusual. If this is your coding style, suppress this message.

501 **Expected signed type** The unary minus operator was applied to an unsigned type. The resulting value is a positive unsigned quantity and may not be what was intended.

502 **Expected unsigned type Unary** ~ being a bit operator would more logically be applied to unsigned quantities rather than signed quantities.

503 **Boolean argument to relational** Normally a relational would not have a Boolean as argument. An example of this is **a < b < c** which is technically legal but does not produce the same result as the mathematical expression which it resembles.

504 **Unusual shift value** Either the quantity being shifted or the amount by which a quantity is to be shifted was derived in an unusual way such as with a bit-wise logical operator, a negation, or with an unparenthesized expression. If the shift value is a compound expression that is not parenthesized, parenthesize it.

505 **Redundant left argument to comma** The left argument to the comma operator had no side effects in its top-most operator and hence is redundant.

506 **Constant value Boolean** A Boolean, i.e., a quantity found in a context that requires a Boolean such as an argument to && or | | or an if() or while() clause or ! was found to be a constant and hence will evaluate the same way each time.

507 **Size incompatibility** A cast was made to an integral quantity from a pointer and according to other information given or implied it would not fit. For example a cast to an unsigned int was specified and information provided by the options indicate that pointers are larger than int's.

508 **extern used with definition** A function definition was accompanied with an extern storage class.

extern is normally used with declarations rather than with definitions. At best the extern is redundant. At worst you may trip up a compiler.

509 **extern used with definition** A data object was defined with a storage class of extern. This is technically legal in ANSI and you may want to suppress this message. However, it can easily trip up a compiler and so the practice is not recommended at this time.

511 **Size incompatibility** A cast was made from an integral type to a pointer and the size of the quantity was too large to fit into the pointer. For example if a long is cast to a pointer and if options indicate that long's are larger than pointers, this warning would be reported.

512 *Symbol* **previously used as static** (*Location*) The *Symbol* name given is a function name that was declared as static in some other module (the location of that declaration is provided). The use of a name as static (i.e., private) in one module and external in another module is legal but suspect.

514 **Unusual use of a Boolean** An argument to an arithmetic operator (+ - / * %) or a bit-wise logical operator (| & ^^) was a Boolean. This can often happen by accident as in:

```
if( flags & 4 == 0 )
```

where the **==,** having higher precedence than&, is done first (to the puzzlement of the programmer).

515 *Symbol* **has arg. count conflict (***Int* **vs.** *Int***) with** *Location* An inconsistency was found in the number of actual arguments provided in a function call and either the number of formal parameters in its definition or the number of actual arguments in some other function call. See the **+fva** option to selectively suppress this message. Also see the appropriate section for information on function prototypes (See Section 7.6 - “Prototype Generation” on page 172.).

516 *Symbol* **has arg. type conflict (no**. I*nt* -- *TypeDiff*) **with** *Location* An inconsistency was found in the type of an actual argument in a function call with either the type of the corresponding formal parameter in the function definition or the type of an actual

argument in another call to the same function or with the type specified for the argument in the function's prototype. The call is not made in the presence of a prototype. See options **-ean**, **-eau**, **-eas** and **-eai** For error inhibition options, see the appropriate section (See Section 3.7 - "Modifying the Report Options" on page 32.). for selective suppression of some kinds of type differences. If the conflict involves types char or short then you may want to consider using the **+fxc** or **+fxs** option (See Section 3.2 - "User Interface" on page 19.).

517 **defined not K&R** The **defined** function (not a K&R construct) was employed and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use defined.

518 **Expected** '(' **sizeof** *type* is not strict C. **sizeof***(type*) or **sizeof** *expression* are both permissible.

519 **Size incompatibility** An attempt was made to cast a pointer to a pointer of unequal size. This could occur for example in a P model where pointers to functions require 4 bytes whereas pointers to data require only 2. This error message can be circumvented by first casting the pointer to an integral quantity (int or long) before casting to a pointer.

520 **Expected void type, assignment, increment or decrement**.The first expression of a for clause should either be an expression yielding the void type or be one of the privileged operators: assignment, increment, or decrement. See also message 522.

521 **Expected void type, assignment, increment or decrement** The third expression of a for clause should either be an expression yielding the void type or be one of the privileged operators: assignment, increment, or decrement. See also message 522.

522 **Expected void type, assignment, increment or decrement** If a statement consists only of an expression, it should either be an expression yielding the void type or be one of the privileged operators: assignment, increment, or decrement. Note that

```
*p++;
```

draws this message but

```
p++;
```

does not. This message is frequently given in cases where a function is called through a pointer and the return value is not **void**.In this case we recommend a cast to **void**. If your compiler does not support the void type then you should use the **-fvo** option.

524 **Loss of precision** *(Context) (Type* **to** *Type)* There is a possible loss of a fraction in converting from a float to an integral quantity. Use of a cast will suppress this message.

525 **Negative indentation from** *Location* The current line was found to be negatively indented(i.e., not indented as much) from the indicated line.The latter corresponds to a clause introducing a controlstructure and statements and other control clauses and braceswithin its scope are expectedto have no less indentation.If tabs within your program are other than 8 blanks you should usethe **-t#** option. See the appropriate section for indentation checking (See Section 7.3 - “Indentation Checking” on page 168.).

526 **Symbol** (*Location*) **not defined**The named external was referenced but not defined and did not appear-declared in any library header file nor did it appear in a LibraryModule. This message is suppressed for unit checkout (**-u** option).Please note that a declaration, even one bearing prototype information isnot 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 filethat you're including. Also make sure that the header file is regarded by *STATIC* as a Library Header file.Alternatively, the symbol may be declaredin a Library Module. See Sthe section on Library Header Files (See Section 3.7.3 - “Library Header File Options” on page 47.).

527 **Unreachable** A portion of the program cannot be reached.

528 **Symbol** *(Location)* **not referenced** The named static variable was not referenced in the module.

529 **Symbol** *(Location)* **not referenced** The named variable was declared but not referenced in a function.

530 **Symbol** *(Location)* **not initialized** An auto variable was used before it was initialized.

531 **Bad field size** The size given for a bit field of a structure exceeds the size of an int .

532 **Return mode of** *Symbol* **inconsistent with** *Location* A declaration (or a definition) of a function implies a differentreturn mode than a previous statement. (The return mode of a function has to do with whether thefunction does, or does not, return a value). A return mode is determined from a declaration by seeing if the function returns void or, optionally, by observing whether an explicit type is given. See the **fdr** flag for a further explanation of this. See also the **fvr** and **fvo** flags for flag options (See Section 3.7.2 - "Flag Options" on page 38.).

533 **Return mode of** *Symbol* **inconsistent with** *Location* A return statement within a function (or lack of a return at the end of the function) implies a different return mode than a previous statement at Location (The return mode of a function has to do with whether the function does, or does not, return a value.) See also the **fvr**, **fvo** and **fdr** flags for flag options (See Section 3.7.2 - "Flag Options" on page 38.).

534 **Return mode of** *Symbol* **inconsistent with** *Location* A call to a function implies a return mode inconsistent with a previous statement. (The return mode of a function has to do with whether the function does, or does not, return a value.) If a call is made just for side effects as, for example, in a statement by itself or the left-hand side of a comma operator, then it is presumed that the function does not return a value. Try: **(void) function();** to call a function and ignore its return value. All other calls presume a returned value. See also the **fvr**, **fvo** and **fdr** flags for flag options (See Section 3.7.2 - "Flag Options" on page 38.).

537 **Repeated include file**: *'FileName'* The file whose inclusion within a module is being requested has already been included in this compilation. The

file is processed normally even if the message is given. If it is your standard practice to repeat included files then simply suppress this message.

538 **Excessive size** The size of an array equals or exceeds 64K bytes.

540 **Excessive size** A string initializer required more space than what was allocated.

541 **Excessive size** The size of a character constant specified with d or h equalled or exceeded **2******b** where b is the number of bits in a byte (established by the **-sb** option) The default is **-sb8**.

542 **Excessive size for bit field** An attempt was made to assign a value into a bit field that appears to be too large to fit. 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.

544 **endif or else not followed by EOL** The preprocessor directive **#endif** should be followed by an end-of-line. Some compilers specifically allow commentary to follow the #endif . If you are following that convention simply turn this error message off.

545 **Suspicious use of &** An attempt was made to take the address of an array name. Since array names are promoted to address, the use of the & is redundant and could be erroneous.

546 **Suspicious use of &** An attempt was made to take the address of a function name. Since names of functions by themselves are promoted to address, the use of the & is redundant and could be erroneous.

547 **Redefinition of symbol** *Symbol* **conflicts with** *Location* The indicated symbol had previously been **#define** d to some other value.

548 **else expected** A construct of the form **if**(*e*); was found which was not followed by an **else**. This is almost certainly an unwanted semi-colon as it inhibits the if from having any effect.

549 **Suspicious cast** A cast was made from a pointer to some enumerated type or from an enumerated type to a pointer. This is probably an error. Check your code and if this is not an error, then cast the item

to an intermediate form (such as an **int** or a **long**) before making the final cast.

550 *Symbol (Location)* **not accessed** A variable (local to some function) was not accessed though the variable was referenced. This could occur for example if 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. Arrays, `struct`s and **union**s are considered accessed if any portion thereof is accessed.

551 *Symbol (Location)* **not accessed** A variable (declared static at the module level) was not accessed though the variable was referenced. See the explanation under message 550 (above) for a description of "access" .

552 *Symbol (Location)* **not accessed** An external variable was not accessed though the variable was referenced. See the explanation under message 550 above for a description of "access".

553 **Undefined preprocessor variable** *Symbol*, **assumed 0** The indicated variable had not previously been defined within a **#define** statement and yet it is being used in a preprocessor condition of the form **#if** or **#elif**. Conventionally all variables in preprocessor expressions should be pre-defined. The value of the variable is assumed to be 0.

555 **#elif not K&R** The **#elif** directive was used and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not use **#elif**.

556 **indented #** A preprocessor directive appeared indented within a line and the K&R preprocessor flag (**+fkp**) was set. Either do not set the flag or do not indent the **#**.

557 **unrecognized format** The format string supplied to **printf**, **fprintf**, **sprintf**, **scanf**, **fscanf**, or **sscanf** was not recognized.

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

559 **size of argument number** *Int* **inconsistent with format** The given argument (to **printf**, **sprintf**, or **fprintf**) was inconsistent with that which was anticipated as the result of analyzing the format string. Argument counts begin at 1 and include file, string and format specifications. For example,

```
sprintf( buffer, %f , 371 )
```

will show an error in argument number 3 because constant 371 is not floating point.

560 **argument no.** *Int* **should be a pointer** The given argument (to scanf, sscanf, or fscanf) should be a pointer. Typically all arguments after the format should be pointers to areas that are to be modified (receive the results of scan ning). Argument counts begin at 1 and include file, string and format specifications. For example

```
scanf( %f , 3.5 )
```

will generate the message that argument no. 2 should be a pointer.

561 **(arg. no.** *Int***) indirect object inconsistent with format** The given argument (to **scanf**, **sscanf**, or **fscanf**) was a pointer to an object that was inconsistent with that which was anticipated as the result of analyzing the format string. Argument counts begin at 1 and include file, string and format specifications. For example if **n** is declared as **int** then:

```
scanf( %c , &n )
```

will elicit this message for argument number 2.

562 **Ellipsis (...) assumed** Within a function proto type a comma was immediately followed by a right parenthesis. This is taken by some compilers to be equivalent to an ellipsis (three dots) and this is what is assumed by *STATIC*. If your compiler does not accept the ellipsis but makes this assumption, then you should suppress this message.

563 **`Label`** *Symbol (Location)* **`not referenced`** The *Symbol* at the cited *Location* appeared as a label but there was no statement that referenced this label.

564 **`variable`** *Symbol* **`depends on order of evaluation`** The named variable was both modified and accessed in the same expression in such a way that the result depends on whether the order of evaluation is left-to-right or right-to-left. One such example is: **`n + n++`** where there is no guarantee that the first access to **`n`** occurs before the increment of **`n`**. Other, more typical cases, are given in the section on **Order of Evaluation**. (See Section 7.1 - "Order of Evaluation" on page 167.)

565 **`tag`** *Symbol* **`not previously seen, assumed file-level scope`** The named tag appeared in a prototype or in an inner block and was not previously seen in an outer (file-level) scope. The ANSI standard is dubious as to how this tag could link up with any other tag. For most compilers this is not an error and you can safely suppress the message. On the other hand, to be strictly in accord with ANSI C you may place a small stub of a declaration earlier in the program. For example:

```
struct name;
```

is sufficient to reserve a place for name in the symbol table at the appropriate level.

566 **`Inconsistent or redundant format char`** *'Char'* This message is given for format specifiers within formats for the *printf/scanf* family of functions. The indicated character found in a format specifier was inconsistent or redundant with an earlier character found in the same format specifier. For example a format containing **`%ls`** will yield this error with the character **`'s'`** indicated.This is because the length modifier is designed to be used with integral or float conversions and has no meaning with the string conversion. Such characters are normally ignored by compilers.

567 **`Expected a numeric field before char`** *'Char'* This message is given for format specifiers within formats for the **`printf/scanf`** family of functions. A numeric field or asterisk was expected at a particular point in the scanning of the format. For

example: **%-d** requests left justification of a decimal integer within a format field. But since no field width is given, the request is meaningless.

568 **unsigned is never less than zero**. Comparisons of the form:

```
u >= 00 <= u
u < 00 > u
```

are suspicious if **u** is an unsigned quantity. This is because unsigned quantities are always greater than or equal to zero. See also message 775.

569 **Loss of information** *(Context)* **(***Int* **bits to** *Int* **bits)** An assignment (or implied assignment, see *Context)* was made from a constant to an integral variable that is not large enough to hold the constant. Examples include placing a hex constant whose bit requirement is such as to require an unsigned int into a variable typed as int. The number of bits given does not count the sign bit.

570 **Loss of sign** *(Context) (Type* **to** *Type)* An assignment (or implied assignment, see *Context*) is being made from a negative constant into an unsigned quantity. Casting the constant to **unsigned** will remove the diagnostic but is this what you want. If you are assigning all 1's to an **unsigned**, remember that **~0** represent all **1**'s and is more portable than **-1**.

571 **Suspicious Cast** Usually this warning is issued for casts of the form:

```
(unsigned) ch
```

where **ch** is declared as **char** and **char** is signed. Although the cast may appear to prevent sign extension of **ch**, it does not. Following the normal promotion rules of C, **ch** is first converted to int which extends the sign and only then is the quantity cast to unsigned. To suppress sign extension you may use:

```
(unsigned char) ch
```

Otherwise, if sign extension is what you want and you just want to suppress the warning in this instance you may use:

```
(unsigned) (int) ch
```

Although these examples have been given in terms of casting a **char** they will also be given whenever this

cast is made upon a signed quantity whose size is less than the casted type. Examples include signed bit fields (a possibility in the new standard), expressions involving **char**, and expressions involving short when this type is smaller than int or a direct cast of an int to an **unsigned long** (if **int**'s are smaller than **long**'s). This message is not issued for constants or for expressions involving bit operations.

572 Excessive shift value 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's are less than 10 bits wide (the usual case). To suppress the message you may cast the shifted quantity to a type whose length is at least the length of the shift value.

573 **Signed-unsigned mix with divide** one of the operands to / or % was signed and the other unsigned; moreover the signed quantity could be negative. For example:

```
u / n
```

where **u** is unsigned and **n** is signed will elicit this message whereas:

```
u / 4
```

will not, even though **4** is nominally an **int**. It is not a good idea to mix unsigned quantities with signed quantities in any case (a 737 will also be issued) but, with division, a negative value can create havoc. For example, the innocent looking:

```
n = n / u
```

will, if **n** is -2 and **u** is 2, not assign -1 to n but will assign some very large value. To resolve this problem, either cast the integer to unsigned if you know it can never be less than zero or cast the unsigned to an integer if you know it can never exceed the maximum integer.

574 **Signed-unsigned mix with relational** The four relational operators are:

```
>  >=  <  <=
```

One of the operands to a relational operator was signed and the other unsigned; also, the signed quantity could be negative. For example:

```
if( u > n ) ...
```

where **u** is unsigned and **n** is signed will elicit this message whereas:

```
if( u > 12 ) ...
```

will not (even though 12 is officially an int it is obvious that it is not negative). It is not a good idea to mix unsigned quantities with signed quantities in any case (a 737 will also be issued) but, with the four relationals, a negative value can produce obscure results. For example, if the conditional:

```
if( n < 0 ) ...
```

is true then the similar appearing:

```
u = 0;
if( n < u ) ...
```

is false because the promotion to unsigned makes **n** very large. To resolve this problem, either cast the integer to unsigned if you know it can never be less than zero or cast the unsigned to an **int** if you know it can never exceed the maximum **int**.

577 **Mixed memory model** (**option** '*String*') The indicated option requested a change to the memory model after part or all of another module was processed. The memory model option should be specified before any module is processed. The most common cause of this error is specifying the memory model after having specified the standard library. This would be a natural error to make if the standard library file were specified via a **LINT** environment variable.

578 **Redefinition of** *Symbol* **hides earlier declaration** (*Location*) A local symbol has the identical name as a global symbol (or possibly another local symbol). This could be dangerous. Was this deliberate? It is usually best to rename the local symbol.

579 **parameter preceding ellipsis has invalid type** When an ellipsis is used, the type preceding the ellipsis should not be a type that would undergo a default promotion such as **char**, **short** or **float**. The

reason is that many compiler's variable argument schemes (using **stdarg.h**) will break down.

580 **Redeclaration causes loss of prototype for** *Symbol (Location)* A declaration of a function within a block hides a declaration in an outer scope in such a way that the inner declaration has no prototype and the outer declaration does. A common misconception is that the resulting declaration is a composite of both declarations but this is only the case when the declarations are in the same scope not within nested scopes. If you don't care about prototypes you may suppress this message. You will still receive other type-difference warnings.

601 **Expected a type, int assumed** A declaration did not have an explicit type. int was assumed. Was this a mistake? This could easily happen if an intended comma was replaced by a semicolon. For example, if instead of typing:

```
doubleradius,
diameter;
```

the programmer had typed:

```
doubleradius;
diameter;
```

this message would be raised.

602 **Comment within comment** The sequence **/*** was found within a comment. Was this deliberate? Or was a comment end inadvertently omitted? If you want *STATIC* 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

```
/*
/*  */
```

then use **-e602**.

603 *Symbol (Location)* **not initialized** The address of the named symbol is being passed to a function where the corresponding parameter is declared as pointer to **const**. This implies that the function will not modify the object. If this is the case then the orig-

inal object should have been initialized sometime earlier.

604 **Returning address of auto** (*Symbol*) The address of the named symbol is being passed back by a function. Since the object is an **auto** and since the duration of an auto is not guaranteed past the **return**, this is most likely an error. You may want to copy the value into a global variable and pass back the address of the global or you might consider having the caller pass an address of one of its own variables to the callee.

605 **Increase in pointer capability** 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.). It 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).

606 **Non-ANSI escape sequence**: *'\String'* - An escape sequence occurred, within a character or string literal, that was not on the approved list which is:

```
\'  \"  \?  \\  \a  \b  \f  \n  \r
\t  \v  \octal-digits  \xhex-digits
```

607 **Parameter substitution** *(Symbol)* **within string**- The indicated name appeared within a string or character literal within a macro and happens to be the same as the name of a formal parameter of the macro as in: #define mac(n) printf(n = %d, , n); Is this a coincidence? The ANSI standard indicates that

the name will not be replaced but since many C compilers do replace such names the construction is suspect. Examine the macro definition and if you do not want substitution, change the name of the parameter. If you do want substitution, set the +fps flag (Parameter within String) and suppress the message.

608 **Assigning to an array parameter** An assignment is being made to a parameter that is typed array.For the purpose of the assignment,the parameter is regarded as a pointer.Normally such parameters are typed as pointers rather than arrays.However if this is your coding style you should suppress this message.

609 **Suspicious pointer conversion** An assignment is being made between two pointers which differ in size(one is **far** and the other is **near**)but which are otherwise compatible.

610 **Suspicious pointer combination** Pointers of different size(one is **far** and the other is **near**)are being compared, subtracted,or paired (in a conditional expression).This is suspicious because normally pointers entering into suchoperations are the same size.

611 **Suspicious cast** Either a pointer to a function is being cast to a pointer to an object or vice versa. This is regarded as questionable by the ANSI standard. If this is not a user error, suppress this warning.

612 **Expected a declarator** A declaration contained just a storage class and a type. This is almost certainly an error since the only time a type without a declarator makes sense is in the case of a `struct` , **union** or `enum` but in that case you wouldn't use a storage class.

614 **auto aggregate initializer not constant** An initializer for an auto aggregate normally consists of a collection of constant-valued expressions. Some compilers may, however, allow variables in this context in which case you may suppress this message.

615 **auto aggregate initializer has side effects** This warning is similar to 614. Auto aggregates (arrays, `struct` s and union s) are normally initialized by a collection of constant-valued expres-

sions without side-effects. A compiler could support side-effects in which case you might want to suppress this message.

616 **`control flows into case/default`** It is possible for flow of control to fall into a **`case`** statement or a **`default`** statement from above. Was this deliberate or did the programmer forget to insert a break statement? If this was deliberate then place a comment immediately before the statement that was flagged as in:

```
case 'a': a = 0;
   /* fall through */
case 'b':  a++;
```

Note that the message will not be given for a case that merely follows another case without an intervening statement. Also, there must actually be a possibility for flow to occur from above.

617 *String* **`is both a module and an include file`** The named file is being used as both an include file and as a module. Was this a mistake? Unlike Error 306 (repeated module) this is just a warning and processing of the file is attempted.

618 **`Storage class specified after a type`** A storage class specifier (**`static`**, **`extern`**, **`typedef`**, **`register`** or **`auto`**) was found after a type was specified. This is legal but deprecated. Either place the storage class specifier before the type or suppress this message.

619 **`Loss of precision`** (*Context*) (**`Pointer to Pointer`**) A far pointer is being assigned to a near pointer either in an assignment statement or an implied assignment such as an initializer, a return statement, or passing an argument in the presence of a prototype (*Context* indicates which). Such assignments are a frequent source of error when the actual segment is not equal to the default data segment. If you are sure that the segment of the far pointer equals the default data segment you should use a cast to suppress this message.

620 **`Suspicious constant (L or one?)`** A constant ended in a lower case letter '**`l`**' . Was this intended to be a one? The two characters look very similar. To avoid misinterpretations, use the upper case letter '**`L`**'.

621 **Identifier clash** (*Symbol* **with** *Symbol* **at** *Location*) The two symbols appeared in the same name space but are identical to within the first count characters set by option **-idlen**(*count,option*) . See **-idlen** for other options (See Section 3.7 - "Modifying the Report Options" on page 38.).

622 **Size of argument no.** *Int* **inconsistent with format** The **Int** 'th argument to **scanf** , **fscanf** or **sscanf** was a pointer whose size did not match the format. For example,

```
int far *p;
scanf( %d , p );
```

will draw this warning (in the default memory model).

623 **redefining the storage class of symbol** *Symbol* **conflicts with** *Location* An inter-module symbol was a typedef symbol in one module and an ordinary symbol in another module. This is legal but potentially confusing. Is this what the programmer intended?

624 **typedef** *Symbol* **redeclared** (*TypeDiff) (Location)* A symbol was **typedef** 'ed differently in two different modules. This is technically legal but is not a wise programming practice.

625 **auto symbol** Symbol has unusual type modifier Some type modifiers such as **far** , **near** , **fortran** are inappropriate for auto variables.

626 **argument no.** *Int* **inconsistent with format** The argument to a **printf** (or **fprintf** or **sprintf**) was inconsistent with the format. Although the size of the quantity was appropriate the type was not. You might consider casting the quantity to the correct type. You could also suppress this message, as more flagrant violations are picked up with warning 559.

627 **(arg. no.** *Int***) indirect object inconsistent with format** The type of an argument to scanf (or **fscanf** or**sscanf**) was inappropriate to the format. However, the argument was a pointer and it pointed to a quantity of the expected size.

628 **no argument information provided for function** *Symbol (Location)* The named function was called but there was no argument information supplied. Argument information can come from a prototype or from a function definition. This usually happens when an old-style function declaration indicates that the function is in a library but no prototype is given for the function nor is any argument information provided in a standard library file. This message is suppressed if you are producing a lint object module because presumably the object module will be compared with a library file at some later time.

629 **static class for function (***Symbol***) is non standard** A static class was found for a function declaration within a function. The static class is only permitted for functions in declarations that have file scope (i.e., outside any function). Either move the declaration outside the function or change static to extern ; if the second choice is made, make sure that a static declaration at file scope also exists before the extern declaration. Though technically the construct is not portable, many compilers do tolerate it. If you suppress the message, *STATIC* will treat it as a proper function declaration.

630 **ambiguous reference to symbol** *Symbol* If the **+fab** flag is set, then if two structures containing the same member name (not necessarily different kinds of structures) are embedded in the same structure and a reference to this member name omits one of the intervening (disambiguating) names, this warning is emitted.

631 **tag** *Symbol* **defined differently at** *Location* The `struct` , union or `enum` tag Symbol was defined differently in different scopes. This is not necessarily an error since C permits the redefinition, but it can be a source of subtle error. It is not generally a programming practice to be recommended.

632 **Assignment to strong type** (*Symbol*) **in context:** *Context* An assignment (or implied assignment, Context indicates which) violates a Strong type check as requested by a **-strong(A** ... option. (See Section 3.7.7 - “Strong Typing Options” on page 61.)

633 **Assignment from a strong type** (*Symbol*) **in context:** *Context* An assignment (or implied assignment, Context indicates which) violates a Strong type check as requested by a **-strong(X** ... option. (See Section 3.7.7 - "Strong Typing Options" on page 61.)

634 **Strong type mismatch** (**type** *Symbol*) in equality or conditional An equality operation (== or !=) or a conditional operation (? :) violates a Strong type check as requested by a **-strong(J** ... option. This message would have been suppressed using flags **Je**. (See Section 3.7.7 - "Strong Typing Options" on page 61.)

635 **resetting strong parent of type *Symbol*, old parent ==** *Symbol* The strong parent of the given Symbol is being reset. This is being done with a **-parent** option or by **typedef** ing one symbol with the other. Note that this may not be an error; you are being alerted to the fact that the old link is being erased. See Section 14.7.7.

636 **ptr to strong type** (*Symbol*) **versus another type** Pointers are being compared and there is a strong type clash below the first level. For example,

```
/*lint -strong(J,INT) */
typedef int INT;
INT *p; int *q;
if( p == q )    /* Warning 636 */
```

will elicit this warning. This message would have been suppressed using strong type flags **Je** or **Jr** or both.

637 **Expected index type** *Symbol* **for strong type** *Symbol* This is the message you receive when an inconsistency with the -index option is recognized. A subscript is not the stipulated type (the first type mentioned in the message) nor equivalent to it within the hierarchy of types. See flag **+fhx** (See Section 3.7.7 - "Strong Typing Options" on page 61.) (See Section 3.7.2 - "Flag Options" on page 38.).

638 **Strong type mismatch for type** *Symbol* **in relational** A relational operation (>= <= > <) violates a Strong type check as requested by a **-strong(J** ... option. This message would have been

suppressed using flags **Jr.** (See Section 3.7.7 - “Strong Typing Options” on page 61.)

639 **Strong type mismatch for type** *Symbol* **in binary operation** A binary operation other than an equality or a relational operation violates a Strong type check as requested by a **-strong(J** ... option. This message would have been suppressed using flag **Jo** . (See Section 3.7.7 - “Strong Typing Options” on page 61.)

640 **Expected strong type** *Symbol* **in Boolean context** A Boolean context expected a type specified by a **-strong(B** ... option. (See Section 3.7.7 - “Strong Typing Options” on page 61.).

641 **Converting enum to int** An enumeration type was used in a context that required a computation such as an argument to an arithmetic operator or was compared with an integral argument. This warning will be suppressed if you use the integer model of enumeration (**+fie**) but you will lose some valuable type-checking in doing so. An intermediate policy is to simply turn off this warning. Assignment of **int** 's to **enum** 's will still be caught.

This warning is not issued for tagless **enum** 's without variables. For example

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

This cannot be used as a separate type. *STATIC* recognizes this and treats **false** and **true** as arithmetic constants.

642 **Format char** *' Char '* **not supported by wsprint**f This means that you are using an option of the form: **-printf(w** ... and you are using a format character not supported by the Microsoft Windows function **wsprintf** . If you are not really using **wsprintf** but are using the **w** flag to get far pointers you should turn this message off.

643 **Loss of precision in pointer cast** A **far** pointer was cast to a **near** pointer. Such casts have had disastrous consequences for Windows programmers. If you really need to make such a cast, you can do it in stages. If you cast to a long first (i.e., some integral type that can hold the pointer) and then into a shorter value, we don't complain.

644 **`Symbol`** (*Location*) **`may not have been initialized`** An auto variable was not necessarily assigned a value before use (See Section 7.11 - "Possibly Uninitialized" on page 180.).

645 **`Symbol`** (*Location*) **`may not have been initialized`** An auto variable was conditionally assigned a value before being passed to a function expecting a pointer to a **`const`** object. See Warning 603 for an explanation of the dangers of such a construct (See Section 7.11 - "Possibly Uninitialized" on page 180.).

646 **`case/default within`** *Kind* **`loop`**; may have been misplaced A case or default statement was found within a **`for`** , **`do`** , or **`while`** loop. Was this intentional? At the very least, this reflects poor programming style.

647 **`Suspicious truncation`** This message is issued when it appears that there may have been an unintended loss of information during an operation involving **`int`** 's or **`unsigned int`** 's the result of which is later converted to long . It is issued only for systems in which **`int`** 's are smaller than long 's. 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 *STATIC* will know you are aware of and approve of the truncation.

648 **Overflow in computing constant for operation:** *String* Arithmetic overflow was detected while computing a constant expression. For example, if **int** 's are 16 bits then 200 * 200 will result in an overflow. String gives the operation that caused the overflow and may be one of: **addition, unsigned addition, multiplication, unsigned multiplication, negation, shift left, unsigned shift left, subtraction,** or **unsigned sub**.

To suppress this message for particular constant operations you may have to supply explicit truncation. For example, if you want to obtain the low order 8 bits of the integer 20000 into the high byte of a 16-bit int , shifting left would cause this warning. However, truncating first and then shifting would be OK. The following code illustrates this where int 's are 16 bits.

```
20000u << 8;                  /* 648 */
(0xFF & 20000u) << 8;         /* OK  */
(unsigned char) 20000u < 8;   /* OK  */
```

649 **Sign fill during constant shift** During the evaluation of a constant expression a negative integer was shifted right, causing sign fill of vacated positions. If this is what is intended, suppress this error, but be aware that sign fill is implementation-dependent.

650 **Constant out of range for operator** *String* In a comparison operator or equality test (or implied equality test as for a case statement), a constant operand is not in the range specified by the other operand. For example, if 300 is compared against a char variable, this warning will be issued. Moreover, if char 's are 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. *STATIC* will take into account the limited precision of some operands such as bit-fields and enumerated types. Also, *STATIC* will take advantage of some computations that limit the precision of an operand. For example,

```
if( (n & 0xFF) >> 4 == 16 ) ...
```

will receive this warning because the left-hand side is limited to 4 bits of precision.

651 **Potentially confusing initializer** An initializer for a complex aggregate is being processed that contains some subaggregates that are bracketed and some that are not. ANSI recommends either minimally bracketed initializers in which there are no interior braces or fully bracketed initializers in which all interior aggregates are bracketed.

652 **#define of symbol** *Symbol* declared **previously at** *Location* A macro is being defined for a symbol that had previously been declared. For example:

```
int n;        #define n N
```

will draw this complaint. Prior symbols checked are local and global variables, functions and **typedef** 'ed symbols, and **struct** , **union** and **enum** tags. Not checked are members of **struct** 's and **union** 's.

653 **Possible loss of fraction** When two integers are divided and assigned to a floating point variable the fraction portion is lost. For example, although double x = 5 / 2; appears to assign 2.5 to x it actually assigns 2.0 . To make sure you don't lose the fraction, cast at least one of the operands to a floating point type. If you really wish to do the truncation, cast the resulting divide to an integral (int or long) before assigning to the floating point variable.

654 **Option** *String* **obsolete; use -width(W ,I)** The option **-w** is now used to set the warning level and should no longer be used to specify the width of error messages. Instead use **-width** with the same arguments as before to set the width. To set the warn-

ing level to 3, for example, use the option **-w3**, not **-w(3)**.

4.7 Informational Messages

701 **Shift left of signed quantity (int)** Shifts are normally accomplished on unsigned operands.

702 **Shift right of signed quantity (int)** Shifts are normally accomplished on unsigned operands. Shifting int 's right is machine dependent (sign fill vs. zero fill).

703 **Shift left of signed quantity (long)** Shifts are normally accomplished on unsigned operands.

704 **Shift right of signed quantity (long)** Shifts are normally accomplished on unsigned operands. Shifting long 's right is machine dependent (sign fill vs. zero fill).

708 **union initialization** there was an attempt to initialize the value of a **union** . This may not be permitted in some older C compilers. This is because of the apparent ambiguity: which member should be initialized. The standard interpretation is to apply the initialization to the first subtype of the **union** .

712 **Loss of precision** (*Context*) (*Type* to *Type*) An assignment (or implied assignment, see *Context*) is being made between two integral quantities in which the first *Type* is larger than the second *Type* . A Cast will suppress this message.

713 **Loss of precision** (*Context*) (*Int* **bits to** *Int* **bits**) 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. The number of bits given does not count the sign bit.

714 *Symbol (Location)* **not referenced** The named external variable was defined but not referenced. This message is suppressed for unit checkout (**-u** option).

715 *Symbol (Location)* **not referenced** The named formal parameter was not referenced.

716 **while(1)** ... A construct of the form **while(1)** ... was found. Whereas this represents a constant in a

context expecting a Boolean, it may reflect a programming policy whereby infinite loops are prefixed with this construct. Hence it is given a separate number and has been placed in the informational category. The more conventional form of infinite loop prefix is **for(;;)** .

717 **do ... while(0)** Whereas this represents a constant in a context expecting a Boolean, this construct is probably a deliberate attempt on the part of the programmer to encapsulate a sequence of statements into a single statement, and so it is given a separate error message. 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);
```

718 *Symbol* **undeclared, assumed to return int** A function was referenced without (or before) it had been declared or defined within the current module. This is not necessarily an error and you may want to suppress such messages (See CHAPTER 11 - Common Problems and Applications" on page 199.). 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.

720 **Boolean test of assignment** An assignment was found in a context that requires a Boolean (such as in an **if()** or **while()** clause or as an operand to **&&** or **||**). This may be legitimate or it could have resulted from a mistaken use of **=** for **==** .

721 **Suspicious use of ;** A semi-colon was found immediately to the right of a right parenthesis in a construct of the form **if(e);** . As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the ' **;** ' is separated by at least one blank from the ' **)** '. Better, place it on a separate line. See also 548.

722 **Suspicious use of ;** A semi-colon was found immediately to the right of a right parenthesis in a

construct of the form **while(e);** or **for**(*e***;** *e***;** *e*); . As such it may be overlooked or confused with the use of semi-colons to terminate statements. The message will be inhibited if the '**;**' is separated by at least one blank from the '**)** '. Better, place it on a separate line.

723 **Suspicious use of =** A preprocessor definition began with an = sign. For example:

```
#define LIMIT = 50
```

Was this intentional? Or was the programmer thinking of assignment when he wrote this?

725 **Expected positive indentation from** *Location* The current line was found to be aligned with, rather than indented with respect to, the indicated line. The indicated line corresponds to a clause introducing a control structure and statements within its scope are expected to be indented with respect to it. If tabs within your program are other than 8 blanks you should use the **-t** option . See the section that describes indentation checking information (See Section 7.3 - "Indentation Checking" on page 168.).

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

727 *Symbol(Location)* **not explicitly initialized** The named static variable (local to a function) was not explicitly initialized prior to use. The following remarks apply to messages 728 and 729 as well as 727. By no explicit initialization we mean that there was no initializer present in the definition of the object, no direct assignment to the object, and no address operator applied to the object or, if the address of the object was taken, it was assigned to a pointer to const . These messages do not necessarily signal errors since the implicit initialization for static variables is 0. However, the messages are helpful in indicating those variables that you had forgotten to initialize to a value. To extract the maximum benefit from the messages we suggest that you employ an explicit initializer for those variables that you want to initialize to 0. For example:

```
static int n = 0;
```

For variables that will be initialized dynamically, do not use an explicit initializer as in:

```
static int m;
```

This message will be given for arrays, **struct** s and **union** s if no member or element has been assigned a value.

728 *Symbol (Location)* **not explicitly initialized** The named intra-module variable (static variable with file scope) was not explicitly initialized. See the comments on message 727 for more details.

729 *Symbol (Location)* **not explicitly initialized** The named inter-module variable (external variable) was not explicitly initialized. See the comments on message 727 for more details. This message is suppressed for unit checkout (**-u**).

730 **Boolean argument to function** A Boolean was used as an argument to a function. Was this intended? Or was the programmer confused by a particularly complex conditional statement. Experienced C programmers often suppress this message.

731 **Boolean argument to equal/not equal** A Boolean was used as an argument to == or != . For example:

```
if( (a > b) == (c > d) ) ...
```

tests to see if the inequalities are of the same value. This could be an error as it is an unusual use of a Boolean (see Warnings 503 and 514) but it may also be deliberate since this is the only way to efficiently achieve equivalence or exclusive or.

Because of this possible use, the construct is given a relatively mild ‘informational' classification. If the Boolean argument is cast to some type, this message is not given.

732 **Loss of sign** (*Context*) (*Type to Type*) An assignment (or implied assignment, see *Context*) is made from a signed quantity to an unsigned quantity. Also, it could not be determined that the signed quantity had no sign. For example:

```
u = n;
u = 4;
```

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.

734 **Loss of precision** (*Context*) (*Int* **bits to** *Int* **bits**) An assignment is being made into an object smaller than an **int** . The information being assigned is derived from another object or combination of objects in such a way that information could potentially be lost. The number of bits given does not count the sign bit. For example if **ch** is a **char** and **n** is an **int** then:

```
ch = n;
```

will trigger this message whereas:

```
ch = n & 1;
```

will not. To suppress the message a cast can be made as in:

```
ch = (char) n;
```

You may receive notices involving multiplication and shift operators with subinteger variables. For example:

```
ch = ch << 2
ch = ch * ch
```

where, for example, ch is an unsigned char . These can be suppressed by using the flag **+fpm** (precision of an operator is bound by the maximum of its operands). See the section on flag options (See Section 3.7.2 - “Flag Options” on page 38.).

735 **Loss of precision** (*Context*) (*Int* **bits to** *Int* **bits**) An assignment (or implied assignment, see Context) is made from a long double to a double . Using a cast will suppress the message. The number of bits includes the sign bit.

736 **Loss of precision** (*Context*) (*Int* **bits to** *Int* **bits**) An assignment (or implied assignment, see Context) is being made to a float from a value or combination of values that appear to have higher precision than a float. You may suppress this message by using a cast. The number of bits includes the sign bit.

737 **Loss of sign in promotion from** *Type* **to** *Type*
An unsigned quantity was joined with a signed quantity in a binary operator (or 2nd and 3rd arguments to the conditional operator **? :**) and the signed quantity is implicitly converted to unsigned. The message will not be given if the signed quantity is an unsigned constant, a Boolean, or an expression involving bit manipulation. For example,

```
u & ~0xFF
```

where **u** is unsigned does not draw the message even though the operand on the right is technically a signed integer constant. It looks enough like an unsigned to warrant not giving the message. This mixed mode operation could also draw Warnings 573 or 574 depending upon which operator is involved. You may suppress the message with a cast but you should first determine whether the signed value could ever be negative or whether the unsigned value can fit within the constraints of a signed quantity.

738 *Symbol (Location)* **not explicitly initialized**
The named static local variable was not initialized before being passed to a function whose corresponding parameter is declared as pointer to const . Is this an error or is the programmer relying on the default initialization of 0 for all static items? By employing an explicit initializer you will suppress this message. See also message numbers 727 and 603.

739 **Trigraph Sequence**' *String* ' **in literal (Quiet Change)** The indicated Trigraph (three-character) sequence was found within a string. This trigraph reduces to a single character according to the ANSI standard. This represents a "Quiet Change" from the past where the sequence was not treated as exceptional. If you had no intention of mapping these characters into a single character you may precede the initial '**?**' with a backslash. If you are aware of the convention and you intend that the Trigraph be converted you should suppress this informational message.

740 **Unusual pointer cast (incompatible indirect types)** A cast is being made to convert one pointer to another such that neither of the pointers is a generic pointer (neither is pointer to **char** , **unsigned char** , or **void**) and the indirect types are

truly different. The message will not be given if the indirect types differ merely in signedness (e.g., pointer to unsigned versus pointer to int) or in qualification (e.g., pointer to **const int** versus pointer to **int**). The message will also not be given if one of the indirect types is a **union** .

The main purpose of this message is to report possible problems for machines in which pointer to **char** is rendered differently from pointer to word. Consider casting a pointer to pointer to **char** to a pointer to pointer to word. The indirect bit pattern remains unchanged.

A second reason is to identify those pointer casts in which the indirect type doesn't seem to have the proper bit pattern such as casting from a pointer to **int** to a pointer to **double** . If you are not interested in running on machines in which **char** pointers are fundamentally different from other pointers then you may want to suppress this message. You can also suppress this message by first casting to **char** pointer or to **void** pointer but this is only recommended if the underlying semantics are right.

741 **Unusual pointer cast (function qualification)** A cast is being made between two pointers such that their indirect types differ in one of the Microsoft qualifiers: **pasca**l , **fortran** , **cdecl** and **interrupt** . If this is not an error you may cast to a more neutral pointer first such as a **void *** .

742 **Multiple character constant** A character constant was found that contained multiple characters, e.g., '**ab**' . This is legal C but the numeric value of the constant is implementation defined. It may be safe to suppress this message because, if more characters are provided than what can fit in an int , message number 25 is given.

743 **Negative character constant** A character constant was specified whose value is some negative integer. For example, on machines where a byte is 8 bits, the character constant ' **xFF**' is flagged because its value (according to the ANSI standard) is -1 (its type is **int**). Note that its value is not **0xFF** .

744 **`switch statement has no default`** A switch statement has no section labeled **`default:`** . Was this an oversight? It is standard practice in many programming groups to always have a **`default:`** case. This can lead to better (and earlier) error detection. One way to suppress this message is by introducing a vacuous **`default: break;`** statement. If you think this adds to much overhead to your program, think again. In all cases tested so far, the introduction of this statement added absolutely nothing to the overall length of code. If you accompany the vacuous statement with a suitable comment, your code will at least be more readable. This message is not given if the control expression is an enumerated type. In this case, all enumerated constants are expected to be represented by **`case`** statements, else 787 will be issued.

745 **`function`** *Symbol* h**`as no explicit type or class, int assumed`** A function declaration or definition contained no explicit type. Was this deliberate? If the flag **`fdr`** (deduce return mode, see the section on flag options) (See Section 3.7.2 - “Flag Options” on page 38.) is turned on, this message is suppressed.

746 **call to** *Symbol* **`not made in the presence of a prototype`** A call to a function is not made in the presence of a prototype. This does not mean that *STATIC* is unaware of any prototype; it means that a prototype is not in a position for a compiler to see it. If you have not adopted a strict prototyping convention you will want to suppress this message with **`-e746`** .

747 **`Significant prototype coercion`** (*Context*) *Type to Type* The type specified in the prototype differed from the type provided as an argument in some significant way. Usually the two types are arithmetic of differing sizes or one is **`float`** and the other integral. This is flagged because if the program were to be translated by a compiler that does not support prototype conversion, the conversion would not be performed. See also Elective Notes 917 and 918.

748 *Symbol (Location)* **`is a register variable used with setjmp`** The named variable is a register variable and is used within a function that calls upon

setjmp . When a subsequent **longjmp** is issued the values of register variables may be unpredictable. If this error is not suppressed for this variable, the variable is marked as uninitialized at this point in the program. More information on messages 749-769 can be found in Section 7.8. page 179 for weak definials information.

749 **local enumeration constant** *Symbol (Location)* **not referenced** A member (name provided as *Symbol*) of an **enum** was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. Compare with messages 754 and 769.

750 **local macro** *Symbol (Location*) not referenced A 'local' macro is one that is not defined in a header file. The macro was not referenced throughout the module in which it is defined.

751 **local typedef** *Symbol (Location)* **not referenced** A 'local' typedef symbol is one that is not defined in any header file. It may have file scope or block scope but it was not used through its scope.

752 **local declarator** *Symbol (Location)* **not referenced** A 'local' declarator symbol is one declared in a declaration which appeared in the module file itself as opposed to a header file. The symbol may have file scope or may have block scope. But it wasn't referenced.

753 **local struct, union or enum tag** *Symbol (Location)* **not referenced** A 'local' tag is one not defined in a header file. Since its definition appeared, why was it not used? Use of a tag is implied by the use of any of its members.

754 **local structure member** *Symbol (Location)* not referenced A member (name provided as *Symbol*) of a **struct** or **union** was defined in a module but was not otherwise used within that module. A 'local' member is one that is not defined in a header file. See message 768.

755 **global macro** *Symbol (Location)* not referenced A 'global' macro is one defined in a header file. This message is given for macros defined in non-library headers. The macro is not used in any of the modules

comprising the program. This message is suppressed for unit checkout (**-u** option). See the section on weak definials information (See Section 7.8 - “Weak Definials” on page 176.).

756 **global typedef** *Symbol (Location)* not referenced This message is given for a typedef symbol declared in a non-library header file. The symbol is not used in any of the modules comprising a program. This message is suppressed for unit checkout (**-u** option).

757 **global declarator** *Symbol (Location)* not referenced This message is given for objects that have been declared in non-library header files and that have not been used in any module comprising the program being checked. The message is suppressed for unit checkout (**-u**).

758 **global struct, union or enum tag** *Symbol (Location)* not referenced This message is given for **struct, union** and **enum** tags that have been defined in non-library header files and that have not been used in any module comprising the program. The message is suppressed for unit checkout (**-u**).

759 **header declaration for** *Symbol (Location)* could be moved from header to module This message is given for declarations, within non-library header files, that are not referenced outside the defining module. Hence, it can be moved inside the module and thereby ‘lighten the load' on all modules using the header. This message is only given when more than one module is being run by *STATIC* .

760 **Redundant macro** *Symbol* **defined identically at** *Location* The given macro was defined earlier (location given) in the same way and is hence redundant.

761 **typedef** *Symbol* **superseded by declaration at** *Location* A **typedef** symbol has been **typedef**’ed earlier at the given location. Although the declarations are consistent you should probably remove the second.

762 **Declaration Symbol superseded by declaration at** *Location* A declaration for the given symbol was found to be consistent with an earlier

declaration in the same scope. This declaration adds nothing new and it can be removed.

763 **Declaration for** *Symbol* **superseded by declaration at** *Location* A tag for a **struct** , **union** or **enum** was defined twice in the same module (consistently). The second one can be removed.

764 **Header file** *FileName* **not directly used in module** *String* The given header file was not used in the given module, however it, itself, included a header file (possibly indirectly) that was used. An example of this is **os2.h** an umbrella header serving only to include other headers. Compare this message with 766.

765 external Symbol (Location) could be made static 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 gooddocumentation. On the other hand, you may want the symbol to remainexternal because debuggers often work only on external names. It's possible, using macros, to have the best of both worlds; see the section on weak definials information (See Section 7.8 - “Weak Definials” on page 176.).

766 **Header file** *FileName* **not used in module** *String* The named header file was not used in processing the named module. It contained no **macro**, **typedef** , **struct** , **union** or **enum** tag or component, or declaration referenced by the module.

767 **macro** *Symbol* **was defined differently in another module** (*Location*) Two macros processed in two different modules had inconsistent definitions.

768 **global struct member** *Symbol (Location)* **not referenced** A member (name provided as *Symbol*) of a **struct** or **union** appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. Since **struct** 's may be replicated in storage, finding an unused member can pay handsome storage dividends. However, many structures merely reflect an agreed-upon convention for accessing storage and for any one program many members are un-

used. In this case, receiving this message can be a nuisance. One convenient way to avoid unwanted messages (other than the usual **-e** and **-esym**) is to always place such structures in library header files. Alternatively, you can place the **struct** within a **++flb ... --flb** sandwich to force it to be considered library.

769 **global enumeration constant** *Symbol (Location)* **not referenced** A member (name provided as *Symbol*) of an **enum** appeared in a non-library header file but was not used in any module comprising the program. This message is suppressed for unit checkout. There are reasons why a programmer may occasionally want to retain an unused **enum** and for this reason this message is distinguished from 768 (unused member). See message 768 for ways of selectively suppressing this message.

770 **tag** *Symbol* **defined identically at** *Location* The **struct** , **union** , or enum tag *Symbol* was defined identically in different scopes. This is not an error but it is not necessarily good programming practice either. It is better to place common definitions of this kind in a header file where they can be shared among several modules. If you do this, you will not get this message. Note that if the tag is defined differently in different scopes, you will receive warning 631 rather than this message.

771 *Symbol (Location)* **conceivably not initialized** The named symbol, declared at *Location* , was initialized in the main portion of a control loop (**while**, **for**, or **do**) and subsequently used outside the loop. If it is possible for the main body of the loop to not be fully executed, then the given symbol would remain uninitialized resulting in an error. *STATIC* does not do a great job of evaluating expressions and hence may not recognize that a loop is executed at least once. This is particularly true after initializing an array. Satisfy yourself that the loop is executed and then suppress the message. You may wish to suppress the message globally with **-e771** or just for specific symbols using **-esym** . Don't forget that a simple assignment statement may be all that's needed to sup-

press the message (See Section 7.11 - “Possibly Uninitialized” on page 180.).

772 *Symbol(Location)* **conceivably not initialized** The address of the named *Symbol* was passed to a function expecting to receive a pointer to a **const** item. This requires the *Symbol* to have been initialized. See Warning 603 for an explanation of the dangers of such a construct. See Informational message 771 for an explanation of "conceivably not initialized".

775 **unsigned quantities cannot be less than zero** An unsigned quantity is being compared for being <=0 . This is a little suspicious since an unsigned quantity can be equal to 0 but never less than 0 . The unsigned 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. See also Warning 568.

776 **Possible truncation of addition** An **int** expression (signed or unsigned) involving addition or subtraction is converted to **long** implicitly or explicitly. Moreover, the precision of a long is greater than that of int. If an overflow occurred, information would be lost. Either cast one of the operands to some form of **long** or cast the result to some form of **int**. See Warning 647 for a further description and an example of this kind of error. See also 790 and 942.

777 **Testing float's for equality** This message is issued when the operands of operators **==** and **!=** are some form of floating type (**float** , **double**, or **long double**). Testing for equality between two floating point quantities is suspect because of round-off error and the lack of perfect representation of fractions. If your numerical algorithm calls for such testing turn the message off. The message is suppressed when one of the operands can be represented exactly, such as 0 or 13.5.

778 Constant expression evaluates to 0 in operation: String A constant expression involving addition, subtraction, multiplication, shifting, or negation resulted in a 0. This could be a purposeful computation but could also have been unintended. If this is intention-

al, suppress the message. If one of the operands is 0 Elective Note 941 may be issued rather than a 778.

779 **String constant in comparison operator:** *Operator* A string constant appeared as an argument to a comparison operator. For example:

```
if( s == abc ) ...
```

This is usually an error. Did the programmer intend to use **strcmp** ? It certainly looks suspicious. At the very least, any such comparison is bound to be machine-dependent. If you cast the string constant, the message is suppressed.

780 **Vacuous array element** A declaration of an array looks suspicious because the array element is an array of 0 dimension. For example:

```
extern int a[][];
extern int a[10][];
```

will both emit this message but

```
extern int a[][10];
```

will not. In the latter case, proper array accessing will take place even though the outermost dimension is missing. If **extern** were omitted, the construct would be given a more serious error message.

781 **Inconsistent use of tag** *Symbol* **conflicts with** *Location* A tag specified as a union , struct , or enum was specified as some other type in another module (location given by *Location*). For example, if tag is specified as **union** in one module and is specified as struct in the current module you will get this message. See also Warning 407.

782 **Line exceeds** *Int* **characters** An internal limit on the size of the input buffer has been reached. The message contains the maximum permissible size. This does not necessarily mean that the input will be processed erroneously. Additional characters will be read on a subsequent read. However the line sequence numbers reported on messages will be incorrect.

783 **Line does not end with new-line** This message is issued when an input line is not terminated by a new-line or when a NUL character appears within an input line. When input lines are read, an **fgets** is

used. A **strlen** call is made to determine the number of characters read. If the new-line character is not seen at the presumed end, this message is issued. If your editor is in the habit of not appending new-lines onto the end of the last line of the file then suppress this message. Otherwise, examine the file for NUL characters and eliminate them.

784 **Nul character truncated from string** During initialization of an array with a string constant there was not enough room to hold the trailing NUL character. For example:

```
char a[3] = abc ;
```

would evoke such a message. This may not be an error since the easiest way to do this initialization is in the manner indicated. It is more convenient than:

```
char a[3] = { 'a', 'b', 'c' };
```

On the other hand, if it really is an error it may be especially difficult to find.

785 **Too few initializers for aggregate** The number of initializers in a brace-enclosed initializer was less than the number of items in the aggregate. Default initialization is taken. An exception is made with the initializer {0} . This is given a separate message number in the Elective Note category (943). It is normally considered to be simply a stylized way of initializing all members to 0.

786 **String concatenation within initializer** Although it is perfectly 'legal' to concatenate string constants within an initializer, this is a frequent source of error. Consider:

```
char *s[] = { abc def };
```

Did the programmer intend to have an array of two strings but forget the comma separator? Or was a single string intended?

787 **enum constant** *Symbol* **not used within switch** A switch expression is an enumerated type and at least one of the enumerated constants was not present as a case label. Moreover, no default case was provided.

788 **enum constant** *Symbol* **not used within defaulted switch** A **switch** expression is an enu-

merated 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`** (*Symbol*) **`to static`** The address of an **`auto`** variable (*Symbol*) is being assigned to a **`static`** variable. This is dangerous because the **`static`** variable will persist after return from the function in which the auto is declared but the **`auto`** will be, in theory, gone. This can prove to be among the hardest bugs to find. If you have one of these, make certain there is no error and use **`-esym`** to suppress the message for a particular variable.

790 **`Suspicious truncation, integral to float.`** This message is issued when it appears that there may have been an unintended loss of information during an operation involving integrals the result of which is later converted to a floating point quantity. The operations that are scrutinized and reported upon by this message are: shift left and multiplication. Addition and subtraction are covered by Elective Note 942. See also 647 and 776.

4.8 Elective Notes

Messages in the 900 level are termed elective because they are not normally on. They must be explicitly turned on with an option of the form **+e9** ... Messages in the range 910-919 involve implicit conversions. Messages in the range 920-930 involve explicit conversions (casts).

911 **Implicit expression promotion from** *Type* **to** *Type* Notes whenever a sub-integer expression such as a **char** , **short** , **enum** , or bit-field is promoted to int for the purpose of participating in some arithmetic operation or function call.

912 **Implicit binary conversion from** *Type* **to** *Type* Notes whenever a binary operation (other than assignment) requires a type balancing. A smaller range type is promoted to a larger range type. For example: 3 + 5.5 will trigger such a message because int is converted to double .

913 **Implicit adjustment of expected argument type from** *Type* **to** *Type* Notes whenever an old-style function definition contains a sub-integer or float type. For example:

```
int f( ch, x ) char ch; float x; { ...
```

contains two 913 adjustments.

914 **Implicit adjustment of function return value from** *Type* **to** *Type* Notes whenever the function return value is implicitly adjusted. This message is given only for functions returning arrays.

915 **Implicit conversion** *(Context) Type* **to** *Type* Notes whenever an assignment, initialization or return implies an arithmetic conversion (Context specifies which).

916 **Implicit pointer assignment conversion** (*Context*) Notes whenever an assignment, initialization or return implies an implicit pointer conversion (*Context* specifies which).

917 **Prototype coercion** (*Context*) *Type* **to** *Type* Notes whenever an implicit arithmetic conversion takes place as the result of a prototype. For example:

```
double sqrt(double);
... sqrt(3); ...
```

will elicit this message because 3 is quietly converted to double .

919 **Implicit conversion** *(Context) Type* **to** *Type* A lower precision quantity was assigned to a higher precision variable as when an int is assigned to a double .

920 **Cast from** *Type* **to void** A cast is being made from the given type to **void** .

921 **Cast from** *Type* **to** *Type* A cast is being made from one integral type to another.

922 **Cast from** *Type* **to** *Type* A cast is being made to or from one of the floating types (**float** , **double** , **long double**).

923 **Cast from** *Type* **to** *Type* A cast is being made either from a pointer to a non-pointer or from a non-pointer to a pointer.

924 **Cast from** *Type* **to** *Type* A cast is being made from a `struct` or a **union** . If the cast is not to a compatible `struct` or **union** error 69 is issued.

925 **Cast from pointer to pointer** A cast is being made to convert one pointer to another such that one of the pointers is a pointer to void . Such conversions are considered harmless and normally do not even need a cast.

926 **Cast from pointer to pointer** A cast is being made to convert a **char** pointer to a **char** pointer (one or both of the **char** s may be unsigned). This is considered a ‘safe' cast.

927 **Cast from pointer to pointer** A cast is being made to convert a **char** (or unsigned **char**) pointer to a non- char pointer. **char** pointers are sometimes implemented differently from other pointers and there could be an information loss in such a conversion.

928 **Cast from pointer to pointer** A cast is being made from a non- char pointer to a char pointer. This is generally considered to be a ‘safe' conversion.

929 **Cast from pointer to pointer** A cast is being made to convert one pointer to another that does not fall into one of the classifications described in 925 through 928 above. This could be nonportable on machines that distinguish between pointer to **char** and pointer to word. Consider casting a pointer to pointer

to **char** to a pointer to pointer to word. The indirect bit pattern remains unchanged.

930 **Cast from** *Type* **to** *Type* A cast is being made to or from an enumeration type.

931 **Both sides have side effects** Indicates when both sides of an expression have side-effects. An example is **n++ + f()** . This is normally benign. The really troublesome cases such as **n++ + n** are caught via Warning 564.

934 **taking address of near auto variable** *(Symbol) (Context)* A source of error in writing DLL libraries is that the stack segment may be different from the data segment. In taking the address of a near data object only the offset is obtained. In supplying the missing segment, the compiler would assume the data segment which could be wrong. See also messages 932 and 933.

935 **int within struct** This Note helps to locate non-portable data items within `struct` 's. If instead of containing int 's and unsigned int 's, a `struct` were to contain short 's and long 's then the data would be more portable across machines and memory models. Note that bit fields and union 's do not get complaints.

936 **old-style function definition for function** *Symbol* An old-style function definition is one in which the types are not included between parentheses. Only names are provided between parentheses with the type information following the right parenthesis. This is the only style allowed by K&R.

937 **old-style function declaration for function** *Symbol* An old-style function declaration is one which does not have type information for its arguments.

938 **parameter** *(Symbol)* **not explicitly declared** In an old-style function definition it is possible to let a function parameter default to int by simply not providing a separate declaration for it.

939 **return type defaults to int for function** *Symbol* A function was declared without an explicit return type. If no explicit storage class is given, then

Informational 745 is also given provided the Deduce Return mode flag (fdr) is on. This is meant to catch all cases.

940 **omitted braces within an initializer** An initializer for a subaggregate does not have braces. For example:

```
int a[2][2] = { 1, 2, 3, 4 };
```

This is legal C but may violate local programming standards. The worst violations are covered by Warning 651.

941 **Result 0 due to operand(s) equaling 0 in operation** ' *String* ' The result of a constant evaluation is 0 owing to one of the operands of a binary operation being 0. This is less severe than Info 778 wherein neither operand is 0. For example, expression (2&1) yields a 778 whereas expression (2&0) yields a 941.

942 **Possibly truncated addition promoted to float** An integral expression (signed or unsigned) involving addition or subtraction is converted to a floating point number. If an overflow occurred, information would be lost. See also messages 647, 776 and 790.

943 **Too few initializers for aggregate** The initializer {0} was used to initializer 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.

950 **Non-ANSI reserved word or construct**: ' *String* ' String is either a reserved word that is non-ANSI or a construct (such as the // form of comment). This Elective Note is enabled automatically by the -A option. If these messages are occurring in a compiler or library header file over which you have no control, you may want to use the option **-elib(950)** . If the reserved word is one which you want to completely disable, then use the option **-rw** (*Word*) .

CHAPTER 5

Libraries

In this chapter you will learn what library modules are, how they are used to describe libraries, how to create a library module, and how to use the alternative library object module.

5.1 Library Modules

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

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

or the equivalent. They are combined with other modules while running STATIC. For example, if sl.c is a Library Module, we can test module.c for conformance by running STATIC on sl.c and module.c.

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

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

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

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

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

Once compilers began introducing prototypes in standard header files, it seemed silly to have a separate set of prototypes in the Library Module and so the Library Modules were modified to merely contain

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

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

But as the size of header files grew, the time to process the Library Module became excessive. Also the processing became somewhat redundant since the header files were being #included in the programmer's own modules. *STATIC* recognizes some headers as library headers(See Section 3.7.2 - “Flag Options” on page 38.). Objects declared within these headers needn't be defined or referenced. If they contain prototypes, then the library is fully described. As a result, for ANSI compilers, our standard library description file has been eliminated. For example, the Microsoft standard library file, *sl-msc.c*, has been replaced by a file containing just options *co-msc.lnt*.

5.1.1 The Current Role of Library Modules

For non-ANSI compilers, the Library Modules serve the same role as ever. For ANSI compilers, they may be used to describe libraries whose header files do not contain prototypes.

5.1.2 Creating a Library Module

Assume you are provided with a graphics library g.lib and a header file g.h describing the library. Ifg.h contains prototypes, you don't need a special Library Module. Just make sure that *g.h* is recognized as a Library Header. (See Section 3.7.2 - “Flag Options” on page 38.).

If **g.h** does not contain prototypes, you can usually prepare them from a textual description of the library provided by the vendor. If you have source for the library, you can generate prototypes using -od option. For example, if files *g1.c* through *g25.c* are the 25 source modules of the library all contained in a single directory, go into that directory and run *STATIC* with the following options: **-u -od**(*gproto.h*). See the section for more information on the **-u** and the **-od** options (See Section 3.7.7 - “Strong Typing Options” on page 61.).

This will output declarations (including prototypes) for all functions and data objects found, to the file *gproto.h*. These declarations will not include struct definitions, however.

5.2 Library Object Modules

If you have source code for a library, an alternative procedure (to producing a library module as in the previous section) is to create a lint object module directly. Assuming we have the same modules g1.c, g2.c, ... g25.c as in the preceding section, create the file *g.lnt* containing:

```
g.lnt
    -u
    -library g1.c
    -library g2.c

.
.
.

    -library g25.c
    -oo(glib.lob)
```

Then run STATIC on g.lnt. The resulting object module, glib.lob, may be used in conjunction with other modules i.e. select glib.lob program.c with the **Load Multiple File** option.

The advantage of this approach is that diagnostic information will be directed to the precise location within the original library source. You may or may not also wish to produce a glib.h file. Note that this method ofusing list objects does not imply that STATIC can likewise implement external script files. However, if you indeed need to use script files, use STW / SMARTS to set up your procedures.

Note that this method of using lint objects does not imply that STATIC can likewise implement external script files. However, if you indeed need to use script files, use STW / SMARTS to set up your procedures

Please also see the appropriate chapter for more information on Lint Object Modules (See CHAPTER 6 - Lint Object Modules" on page 161.).

CHAPTER 6

Lint Object Modules

This chapter defines Lint Object Modules, how they are used, and how to produce one.

6.1 What is a Lint Object Module (LOB)?

Note: Lint Object Modules are recommended for large programs consisting of 25,000 lines (1 million bytes) or more of code. If your programs are more modest, you may safely skip this chapter.

A *Lint Object Module* is a summary (in binary form) of all the external information within a C module (or modules). *STATIC* can then use this information to compare with other modules for consistency. For example, if module **`alpha.c`** consists of:

```
alpha.c
   void beta(x)
double x;
{
gamma(3);
}
```

then the associated object module for alpha.c (call it alpha.lob) will contain information that beta was defined with a double argument returning void and gamma was called with an int constant argument. The object file will retain the name of the original module, line number information and the names of all included header files.

6.2 Why are LOBs Used?

Lint Object Modules are used to speed up the processing of multi-module programs. Consider the figure on the following page which shows a program consisting of 9 modules **`a1.c`** through **`a9.c`**. Rather than running *STATIC* on all the source modules together, the programmer has run *STATIC* on the modules separately producing a Lint Object Module for each source module. A typical command might be using the options **`-u`**

and **-oo** together. See the correct section for more information on these options (See Section 1.2 - “Language Definition” on page 3.).

This produces **a1.lob**. The **-u** (unit checkout) option should always be used when producing a Lint Object Module. All the usual messages will be produced, appropriate to unit checkout.

(You may need the option **-zero** or **-zero(500)** to insure producing the object module in spite of error messages.) See the correct section for more information on the **-zero** option (See Section 1.2 - “Language Definition” on page 3.).

After the Lint Object Modules are produced, they must then be run together to make sure they are all consistent with one another. This is also shown in the figure. This can be done by running *STATIC* on ***.lob”**

This produces the inter-module messages. If a single change is made to any source module, say to a**1.c** then only one object module needs to be regenerated. This is then combined with all of the other Lint Object Modules. The time required to process the collection of Lint Object Modules is typically short, on the order of processing just one source module and so the time savings is substantial. The observant reader will note that this process lends itself to incremental staticing through a `make` facility. This is discussed later.

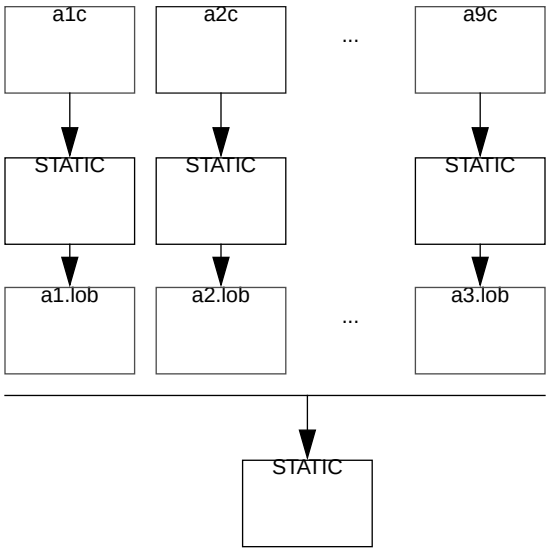

FIGURE 33 LOB

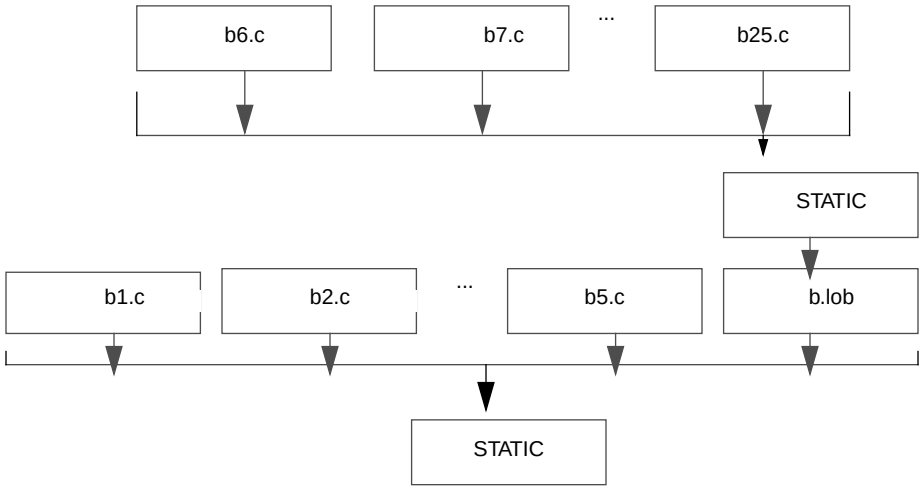

FIGURE 34 LOB-2

Another way to use Lint Object Modules is shown in the figure (See Figure 34 "LOB-2" on page 163.). Here a project consists of modules **`b1.c`** through **`b25.c`**. We assume our programmer is only responsible for modules **`b1.c`** through **`b5.c`** with other members of a team responsible for other modules. Accordingly a summary of the external information of modules b6.c through **`b25.c`** is captured in the Lint Object Module **`b.lob`**.

This is then used when running **`b1.c`** through **`b5.c`** as is shown in the figure (See Figure 34 "LOB-2" on page 163.). This dramatically improves the speed of running *STATIC*. It is instructive to compare this approach with that of producing function prototypes for all the functions in **`b6.c`** through **`b25.c`**. Function prototypes can be produced with the **`-od`** option (output declarations). Function prototypes do not contain information such as what line of what file contains information inconsistent with another file. It does not indicate which variables have been initialized or accessed or which objects have been referenced. The information in **`.lob`** files is, therefore, more complete and indicative than prototype information, as well as quite fast.

6.3 Producing a LOB

The option **-oo** [(*filename*)] (See Section 1.2 - “Language Definition” on page 3.) will cause binary information for all modules on the command line to be output to the named file. The “**oo** “ stands for “output object”. If *filename* is omitted, as in the option:

```
-oo
```

then a name is formed from the first module name using an extension of “**.lob** “ (a name ending in “**.lob**” is recommended since *STATIC* uses this extension on input to determine that the file is an object module and not a source module). For example:

```
lint -u alpha.c -oo
```

will output binary external information about alpha.c into the file alpha.lob.

6.4 Make Files

Lint Object Modules are well adapted for use with a `make` facility. For example, a `make` script which follows the Unix `make` conventions can be of the following form. (Note: if you are following the Microsoft `make` conventions place the directive starting with “project.lob” at the end of the script).

```
 c.lob:
    llint -u make.lnt $* -oo

project.lob: module1.lob module2.lob
module3.lob
    llint make.lnt *.lob

module1.lob: module1.c

module2.lob: module2.c

module3.lob: module3.c



where make.lnt contains
  6

make.lnt
    -os(temp) +vm std.lnt
```

Here a program consists of three modules: **module1.c**, **module2.c** and **module3.c**.

If any of these modules is altered, the command on the 2nd line is executed. If no flaws are found, the option **-oo** will cause object module **i.lob** to be written.

The second request within the make file seems to suggest that project.lob is created during that step. project.lob is a fictitious name which forces this command to take place. We could have produced a **project.lob** with **-oo(project.lob)** but it wouldn't be particularly useful. The file **make.lnt** houses options used for running *STATIC* within a make file. The **-os** (*filename*) option (See Section 3.7.8 - "Other Options" on page 78.) has the effect of redirecting the messages (much as filename). Unlike redirection, the option can be placed within an indirect file as shown here.

A Lint Object Module will normally not be produced if as much as one error message is produced. If you want the make script to go on in spite of some messages, then you may use the option: **-zero** to force an exit code of zero. You probably don't want to use the **-os()** option in this case since your messages will be overwritten by the next command. Use >>temp on the command line instead. Alternatively, you may want to parameterize the **-zero** option. An option of **-zero(2n)**, as in **-zero(700)**, will have the effect of not counting messages whose message number is equal to or higher than the specified number *n*, 700 in this example. This is much like a compiler producing warnings but going on to produce an object module as well.

6.5 Library Modules

Library modules (See Section 5.1 - "Library Modules" on page 157.) are used to describe libraries and are usually for non-ANSI compilers. They contain the option **/*lint -library*/** or its equivalent. These can also be in object form. For compilers that support ANSI prototypes, library modules are becoming obsolete because a header file (or files) describing the library is generally all that *STATIC* needs to determine whether function calls are compatible with a library. For compilers that do not support prototypes, it is necessary to have an extra module that describes arguments to library functions. Call this sl.c. For large library modules it makes sense to produce an object version of this module. The command **llint co.lnt sl.c -oo** (where **co.lnt** is a compiler options file) will produce the file **sl.lob**. Declarations of objects that are not referenced are not normally a part of a Lint Object Module. But, in

this case, where the only input modules are library modules, an exception to this rule is made. For example suppose **sl.c** contains:

```
 sl.c
/*lint -library */

double sin(double);
```

Since **sin** is neither defined nor referenced within the module (it is only declared) it would not normally be retained in an object module. But because this is a library module and because there are no non-library modules being presented to *STATIC*, all library declarations are retained.

6.6 Options for LOB's

To conserve on space, Lint Object Modules do not, by default, contain objects that have merely been declared (but not referenced or defined). The option **fod** (Object module receives all Declarations) overrides this default behavior. Also, by default, library objects unless referenced or defined are not normally included, again to save space. **fol** forces all library symbols to be included in the module. This option is not normally needed because when making an object module from only library modules, the flag is automatically thrown on. (See Section 5.2 - "Library Object Modules on page 159.).

6.7 Limitations of LOB's

To conserve on space, macros are not placed within Lint Object Modules. This affects Informational messages 755 (global macro not referenced) and 767 (macro was defined differently in another module). For this reason you will occasionally want to run *STATIC* on all your source files together even though your normal modus operandi is to use Lint Object Modules.

CHAPTER 7

Special Features

This chapter discusses how *STATIC* checks for the following: out-of-order expressions, formats, indentations, consts, and volatiles. It also discusses prototype generation, header file regeneration, parameter matching, weak definials, global variables, and function mimicry.

7.1 Order of Evaluation

Expressions whose value depends on the order-of-evaluation are flagged with Warning 564.This is a very infamous problem with C but very few compilers will diagnose it. In general, the compiler is not obligated to evaluate expressions left-to-right or indeed in any particular order. For example,

```
n++ + n
```

is ambiguous; if the left hand side of the binary + operator is evaluated first, the expression will be one greater than if the right hand side is evaluated first. Some other, more common examples are:

```
a[i] = i++;
f( i++, n + i );
```

In the first case, it looks as though the increment should take place after computing the array index. But, if the right hand side of the assignment operator is evaluated before the left hand side, the increment is done before the index is computed. Although assignment looks as though it should imply an order of evaluation it does not. The second example is ambiguous because the order of evaluation of arguments to a function is not guaranteed. The only operators that imply an order of evaluation is Boolean AND (&&), Boolean OR (| |), conditional evaluation (?:), and the comma operator (,). Hence:

```
if( (n = f()) && n > 10 ) ...
```

works as expected and you don't get a warning, whereas:

```
if( (n = f()) & n > 10 ) ...
```

will elicit a message.

In general, for every binary operator that does not have an implied order of evaluation, the set of variables modified by each side is compared with the set of variables accessed by the other side and a warning is issued for each common member. A variable is considered modified if it is subject to autoincrement or autodecrement or is assigned to.

7.2 Complete Format Checking

STATIC completely checks for printf and scanf (and family) format incompatibilities. For example,

```
printf( "%+c", ... )
```

will draw a warning (566) because the plus flag is only useful for numeric conversions. There are over a hundred such combinations that will draw this warning and compilers do not normally flag the inconsistencies. Other warnings that complain about bad formats are 557 and 567. We follow the formatting rules established by ANSI C.

Perhaps more importantly we also flag arguments whose size is inconsistent with some format (Warnings 558, 559, 560 and 561). Thus, with %d format, both integers and unsigned int are allowed but not double and not long if these are larger than int. Similarly, scanf type formats require that the arguments be pointers to objects of the appropriate size. If only the type of the argument (but not its size) is inconsistent with the format character, Warnings 626 or 627 will be given.

The -printf and -scanf options allow a user to specify functions that resemble a member of the printf or scanf family.

7.3 Indentation Checking

Indentation checking can be used to locate the origins of missing left and right braces. It can also locate potential problems in a syntactically correct program. For example, consider the code fragment:

```
if( ... )
if( ... )
statement
else statement
```

Apparently the programmer thought that the else associates with the first if whereas a compiler will, without complaint, associate the else with the second if. *STATIC* will signal that the else is negatively indented with respect to the second if.

There are two forms of messages; Informational 725 is issued in the case where there is no indentation (no positive indentation) when indentation

is expected and Warning 525 is issued when a construct is indented less than (negatively indented from) a controlling clause. 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 (See Section 3.7.8 - "Other Options" on page 78.).

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 *STATIC* 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 il flag is given (See Section 3.7 - “Modifying the Report Options” on page 32.) in which case it undergoes a weak check.

7.4 const Checking

const is fully supported. We recommend that you incorporate the use of const in your programming style as there are several unexpected benefits from using this new keyword. Consider the program fragment:

```
char *strcpy( char *, const char * );
const char c = 'a';
const char *p = &c;

void main()
    {
    char buf[100];

    c = 'b';
    *p = 'c';
    strcpy( p, buf );
.
.
```

.

This will draw four separate messages. Clearly c and *p, since they are const should not be modified (Error 111). Also, passing p as a first argument to strcpy draws a warning (605) because of an increase in pointer capability. Finally, passing buf as the second argument draws a warning (603) because buf hadn't been initialized and a function expecting a pointer to a const value will not do the initialization. If your compiler does not support const you may wish to use:

```
#ifdef _lint
#define CONST const
#else
#define CONST
#endif
```

at the head of your source code. Then use CONST rather than const throughout.

7.5 volatile Checking

volatile has only modest error checking properties. A variable declared volatile should not be used twice in the same expression (order of evaluation problems) and declarations containing the keyword are checked for consistency. Pointers declared as pointing indirectly to volatile objects may not be used indirectly twice in the same expression and functions declared to return volatile values may not be called twice in the same expression. (They receive Warning 564). For example:

```
volatile char *p;
volatile char f();

n = (f() << 8) | f();     /* Warning 564 */
n = (*p << 8) | *p;       /* Warning 564 */
```

The reason the warning is given is that it is presumed that each access of a volatile object or function produces a potentially different value and that the order of evaluation cannot be guaranteed.

You may declare functions to be volatile if they have side effects such as returning the next character of an input stream or global string. A pointer may be pointing to a volatile object if it is used for memory mapped I/O. If your compiler doesn't support volatile, you may want to hide this from your compiler using the method described for const (See Section 7.4 - “const Checking” on page 170.).

7.6 Prototype Generation

The option:

```
-od[s][i][f][width](filename)
```

outputs declarations to `filename` and is frequently employed to generate a set of prototypes for the functions defined within a module. Please also refer to the correct section for further information on this option (See Section 3.7 - “Modifying the Report Options” on page 32.). For example using the -od(alpha.h) on alpha.c produces a header file to be #included by routines that use the services provided by alpha.c.

A prototype will be generated for functions defined “old-style” as in:

```
int f(x) int x; {return x;}
```

as well as for functions defined “new-style” as in

```
int f(int x) {return x;}
```

The same prototype, i.e.,

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

is generated in each case. If there is a clash between the declaration and the definition, the definition wins. This is to make it possible to regenerate header files . If the variable-arguments flag

```
+fva
```

has been set for the function when declared or defined, then no list of parameter types is generated. Thus:

```
/*lint +fva  */
int f();
/*lint -fva  */

int f(int x) {return x;}
```

results in:

```
int f();
```

being generated. If a limit on the number of arguments to be checked is provided as in:

```
/*lint +fva1  */
int g();
/*lint -fva  */

int g(x,y) int x,y; {return x+y;}
```

then the output of the -od file will contain:

```
int g(x,...);
```

7.6.1 Header File Regeneration

We have designed the generation of declarations (`-od`) in such a way that header files can be regenerated after a modification is made to the original module. For example, assume module alpha.c is part of a larger project. To generate a header file for alpha use the following option on alpha.c:

```
-u -od(alpha.h)
```

and then include alpha.h in every module using the facilities provided by alpha.c as well as alpha.c itself. If we change a definition within alpha.c the clash between the header file and the module will be noticed by either your compiler or *STATIC*.

Although messages will be issued reporting the inconsistencies, alpha.h will be rewritten using the information from the definitions within alpha.c rather than the declarations within alpha.h. To confirm this, run *STATIC* on alpha.c with only the `-u` option.

7.6.2 -odi (static functions)

Prototypes for static functions (i.e., functions with internal linkage) are not automatically generated with -od. This is consistent with the idea that prototype output is intended for inter-module communication. To get prototypes for static (internal) functions, as well as external objects, use: -odi(filename).

7.6.3 -odf (only functions)

If `f` is specified as in `-odf`, output is limited to functions (no data declarations).

7.6.4 -ods (structs)

Consistent with the idea that you are starting with a program that already is properly header'ed, so that different modules already "see" all the structs, unions and enums that they need, we do not normally generate definitions of these objects as part of -od, If you want them, use -ods(filename).

7.6.5 -odwidth

Prototypes are broken (with a new-line character) after spaces, commas and semicolons whenever the current width exceeds the specified width (default width is 66).

7.6.6 Precautions with Prototypes

Prototypes do not play just the passive role of enabling compilers and lint processors to more intelligently diagnose errors. They also play an active role in silently converting arguments. Message 747 will detect flagrant conversions. You may also want to detect subtle conversions by turning on Elective Notes 917 and 918 (with the options +e917 +e918).

7.6.7 typedef Types in Prototypes

It is possible to produce prototypes containing typedef names. See Section 3.7.7 on page 61.

7.7 Exact Parameter Matching

Types of function parameters are not always taken literally. If a parameter is typed array, for example, this is considered a stylistic way of indicating pointer. With old-style function definitions, parameters of type char, unsigned char, short, unsigned short, and float are quietly promoted for the purpose of matching up with arguments. (However, for subsequent use within the function the original type is used). For example:

```
int f( ch, sh, fl, a )
char ch;
short sh;
float fl;
int a[10];
{
. . .
```

is the start of an old-style function definition (i.e., a definition that does not place type information between the parentheses). For the purpose of detecting type conflict, *STATIC* will promote the types of `ch` and `sh` to

`int`, `fl` to `double`, and `a` to `pointer` to `int`. If, for example, a `char` argument is passed as the first argument to `f()` this argument is also promoted by the rules of C to type `int` so that no mismatch is reported. But it can be argued that some valuable type information is lost in this way. If an `int` is accidentally passed as first argument to `f()` the mismatch would go unreported. For this reason several flags are available to inhibit the usual promotion rules for parameters of this type:

```
+fxa  eXact Array matching
+fxc  eXact Char matching
+fxf  eXact Float matching
+fxs  eXact Short matching
```

These flags are effective only if the formal parameter is the one that is normally promoted. Consider:

```
char ch;
int g(i)
int i;
{  . . .  }
    . . .  g( ch );
```

Here, the actual argument `ch` is considered matched against the formal parameter i even if the `fxc` flag is set. On the other hand passing an int to the first argument of `f()` (previous example) would be flagged. (In a beta release we gave a message in both instances but this rendered the flag almost useless). With exact array matching, for example, only an array of 10 int's may be passed as fourth argument to function `f()` (previous example). Pointers may not be passed to array parameters but, as indicated above, array arguments may be passed to pointer parameters.

With `char` and short exact matching, the argument must be the exact type declared or a compatible constant. However, if the argument is an expression involving an operation other than the conditional (?:), the operation is assumed to be carried out with at least `int` precision. It will not match a `char` or `short` parameter. To be compatible with a parameter typed char or short, constants need to be able to fit within the type without loss of precision. For example, 0 is compatible with `char` and `short` as well as with `int`.

When *STATIC* encounters calls to a function prior to seeing a definition (or a prototype) there is a slight problem. For example, if it sees:

```
f( 513, 'a' );
. . .
f( 'b', 814 );
```

Then what should *STATIC* record with regard to the arguments being passed to `f()`? Remember that no error should be reported if the param-

eters are subsequently discovered to be typed int. A worst case argument is saved, i.e., one that will match the fewest subsequent types. In this case it will be recorded that `f()` had been passed two `int`'s. If the definition:

```
void f( i, c )
int i;
char c;
{  . . .  }
```

is later discovered, then a mismatch is reported for the 2nd parameter. Unfortunately the position information of the offending argument will be lost because the information about the arguments had been derived from two different places. You will see "location unknown" in the message. If you can't find the position by just searching, reorder the modules so that the definition appears first. It may be more convenient to place a truncated definition in a dummy module before all the other modules. The `fxc` and `fxs` options may be useful if you are matching old-style function definitions with new-style prototypes. For example:

```
void g(char);

void g(c) char c; {  . . .  }
```

is considered erroneous by *STATIC* since by the rules of ANSI the first `char` does not get promoted but the second one does. On the other hand, if the second char is changed to read `int` then the Microsoft compiler reports a type mismatch. Perhaps this is fair since ANSI C considers both sequences to be erroneous since a new-style prototype is being mixed with an old-style definition. A way to get around this difficulty and still retain the old-new confrontation is to use the `fxc`.

Note: Although *STATIC* does not complain about the argument difference it will complain because `g()` is retyped (type difference = promotion). To get *STATIC* to completely ignore this, it is necessary to also use the option: `-etd(promotion)`.

With float exact matching the considerations are similar to the case of char and short exact matching. Only constants that are `float` (such as 1.2f) are considered compatible with a float parameter. The previously cited *STATIC* Microsoft conflict does not occur with the float type so that the use of `fxf` may not be as compelling as with the other flags.

7.8 Weak Definials

The weak definials consist of the following:

- macro definitions

- typedef's
- declarations
- struct, union and enum definitions and members

They are compile-time entities and for this reason, perhaps, they are not used as carefully or as scrupulously as run-time objects. Their definitions may be redundant or may lay around unused. Sometimes they are defined inconsistently across modules. Because they are only compile-time entities, they are referred to as weak. The word definial means simply that which is defined. It has the benefit of no prior use and hence semantic neutrality in C. Where there is no possibility of confusion, we will use the word definial as an abbreviation for the term weak definial.

The weak definials are important because they represent those entities normally placed into header files to provide communication for the many modules that comprise a program. To determine whether a header file is unused or not depends upon whether any of its weak definials have been used.

Informational messages in the range 749-769 are reserved for the weak definials. *STATIC* is able to report on unused header files, (764 and 766), definials within (non-library) header files that are not used, (755-758, 768), non-header definials that are not used (750-573), redundant definials (760-763) and conflicting definials. If you run *STATIC* on some previously untouched source code, you may well want to turn these messages off. However, if you just want to see header anomalies, you might want to try:

```
llint -w1 +e749 +e75? +e76?   ...
```

Whether a header file is used or not depends on whether any of its definials have been used by any other file. The operative word here is 'other'. For example, let the complete contents of hdr.h be:

```
hdr.h
typedef int INT;
extern INT f();
```

Assume a single module includes this header file but makes no use of either f or of INT. The definial INT would be considered used by virtue of its appearance within the declaration of f and f would be reported unused. The header file would be reported unused by the module because the only use of any of its definials was a self reference, a reference to INT from within the same header file. If the declaration of f were removed, then INT would be reported as unused and hdr.h would also be reported as unused. Consider the following example:

```
hdr.h
typedef int INT;
alpha.c
#include "hdr.h"
typedef int INT;
INT x = 0;
```

Is the definial INT within hdr.h being used or not? Is the header file hdr.h being used? Since we have two identical declarations for INT it is hard to say. What we do in this case is report that the second typedef is redundant. We then act as if the second never appeared and so the header file appears to have been used. If the second typedef were a different type, an error would be reported, and the first typedef would be considered unused.

A special message (759) is issued for objects declared in headers but then not referenced outside the module that defines them. (This message is automatically suppressed if there is only one module being processed). If a declaration is used by only one module, it can be removed from the header file thereby reducing its size. Header files have a tendency to become big and fat; compilers are always indicating when something has to be added but hardly ever indicate when something can be deleted; this produces unidirectional growth. Message 759 is intended to combat this tendency. A related message (765) is the identification of all objects that are ‘file-scopable'; i.e., external objects that may be tagged static and hence not placed into the pool of external names. A programmer may not at all be interested in staticizing everything because modern debuggers sometimes depend critically on such external symbols. However you may wish to employ the following technique:

```
#if debug && !defined(_lint)
#define LOCALF
#define LOCALD extern
#else
#define LOCALF static
#define LOCALD static
#endif
```

LOCALD stands for local Declaration.

LOCALF stands for local deFinition.

These are used as:

```
LOCALD double func( );
```

```
.
.
.
LOCALF double func( ) {return 37.5; }
```

For debugging (and provided we are not running *STATIC*)

the function `func` is external and its name is available to the debugger. Otherwise it is made static. The macros provide good documentation and *STATIC* enforces compliance.

7.9 UNIX Lint Options

The following options are available for compatibility with other `lint`'s, in particular with the original UNIX `lint`. They may be embedded within C code, where they appear as a comment. They are all of the form:

```
/* Optional-blanks Keyword Optional-
blanks */
```

LINT LIBRARY — This is equivalent to `/*lint -library */`. (See Section 5.2 - "Library Object Modules" on page 159.)

ARGUSED — Inhibits complaints about function parameters not being used for the duration of a single function. It is placed just before a function definition. It is equivalent to: `/*lint -e715 */` placed before a function definition and restored with `/*lint -restore */` afterward.

VARARGS [N] — When this option is placed before a function declaration (or definition) it has the effect of `/*lint +fva[N] */` for just one function. Like `ARGSUSED`, it has an automatic reset feature.

NOTREACHED — This option is equivalent to `/*lint -unreachable */`

NOSTRICT — This inhibits certain kinds of strict type-checking for the next expression. The type differences that are relaxed are those denoted as nominal, signed/unsigned, ellipsis, promotion and ptrs to incompatible types.

The equivalent *STATIC* option is:

```
/*lint -etd(nominal,signed/unsigned,ellip-
sis)
     -etd(promotion,"ptrs to incompatible
types") */
... expression...
```

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

(See TypeDiff in Chapter 11).

7.10 Static Initialization

Traditional lint processors do not flag uninitialized static (or global) variables because the C language defines them to be 0 if no explicit initialization is given. But uninitialized statics, because they can cover such a large.ix uninitialized statics scope, can be easily overlooked and can be a serious source of error. *STATIC* will flag static variables that have no initializer and that are assigned no value. For example, consider:

```
int n;
int m=0;
```

There is no real difference between the declarations as far as the C language is concerned but *STATIC* regards m as being initialized and n not initialized. If n is nowhere assigned a value, a complaint will be emitted.

7.11 Possibly Uninitialized

This section has to do with messages 644, 645 (“may not have been initialized”) and 771, 772 (“conceivably not initialized”), and 530 (“not initialized”).

STATIC we take into account flow-of-control in our “not initialized” messages. For example:

```
if( a ) b = 6;
else c = b;
a = c;
```

assume that neither `b` nor `c` were previously initialized. *STATIC* reports that b is not initialized (when its value is assigned to c) and that c may not have been initialized (when its value is assigned to a). In earlier versions (and in conventional lint's) a single unintelligent sweep is taken which would regard `b` and `c` as having been initialized prior to use.

`while` loops and `for` loops are not quite the same as if statements. Consider, for example, the following code:

```
while ( n-- )
{
b = 6;
.
.
.
}
c = b;
```

assuming that b had not earlier been initialized, we report that b is "conceivably not initialized" when assigned to c and give a lighter Informational classification. The reason for distinguishing this case from the earlier one is that it could be that the programmer knows the body of the loop is always taken at least once. By contrast, in the earlier case involving if statements, the programmer would be hard-pressed to say that the if condition is always taken, for that would imply, at the least, some redundant code which could be eliminated.

The switch is more like an `if` then a `while`. For example:

```
switch ( k )
{
   case 1:  b = 2; break;
   case 2:  b = 3;
   /* Fall Through */
   case 3:  a = 4; break;
   default:  error();
}
c = b;
```

Although `b` has been assigned a value in two different places, there are paths that might result in `b` not being initialized. Thus, when `b` is assigned to c a possibly-uninitialized message is obtained. To fix things up you could assign a default value to b before the switch. This quiets *STATIC* but then you lose the initialization detection in the event of subsequent modifications. A better approach may be to fix up the case's for which b has not been assigned a value. We will show this below. If the invocation of `error()` is one of those instances which "can't occur but I'll report it anyway," then you should let *STATIC* know that this section of code is not reachable. If `error()` does not return, it should be marked as not returning by using the option `-function(exit,error).`

This transfers the special property of the exit function to error. Alternatively, you may mark the return point as unreachable as shown in the following fixed up example:

```
switch (k)
{
   case 1:  b = 2; break;
   case 2:
   case 3:  b = 3; a = 4; break;
   default:  error();
   /*lint -unreachable */
}
c = b;
```

Don't make the mistake of placing the `-unreachable` before the call to error() as this property is not transmitted across the call. If there is a break after the call, make sure the directive is placed before the break. Code after a break, is never considered reachable, so the directive placed after the break would have no effect. Another way to get the “not initialized” message is to pass a pointer variable to free (or to some function like free (See Section 7.12 - “Function Mimicry (-function)” on page 183.). For example:

```
if(n) free( p );
.
.
.
p->value = 3;
```

will result in p being considered as possibly not initialized at the point of access. Forward `goto`'s are supported in the sense that the initialization state of the `goto` is merged with that of the label. Thus, if b is not yet initialized, the code:

```
if (a) goto label;
b = 0;
label: c = b;
```

will receive a possibly-uninitialized message when `b` is assigned to `c`. However, backward goto's, since they do not reduce the initialization state, are ignored. When checking for possibly uninitialized variables is first applied to a large mature project, there will be a small number of false hits. Experience indicates that they usually involve code that is not especially well structured or may involve some variation of the following construct:

```
if(x)  initialize y
.
.
.
if(x)  use y
```

For these cases simply add an initializer to the declaration for y or use the option -esym(644,y).

7.12 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**.

```
-function(Function0, Function1[,
Function2]...)
```

specifies that Function1, Function2, etc. are like Function0 in that they exhibit special properties normally associated with Function0. The special functions (Function0) are as follows:

abort This is treated like **exit** below.

exit Statements immediately following a call to **exit** are considered unreachable.

free The first argument is a pointer which is subsequently regarded as uninitialized.

longjmp This is like **exit**.

realloc This is similar to **free**; the first argument is a pointer which is considered possibly freed and hence possibly uninitialized.

For example, if you have a function called my **free** that disposes of the storage associated with a pointer, you may use the following option:

```
-function(free,myfree)
```

Then the following code sequence will draw a complaint:

```
.
.
.
myfree(p);
x = p->y;   /* not initialized */
.
.
.
```

If the argument of your free function that bears a pointer to be freed is not the first, you may use functional notation to indicate which argument applies. For example if free2 frees two pointers (first and second arguments) you may use the pair of options:

```
-function(free,free2(1))
-function(free,free2(2))
```

or combine them into a single option:

```
-function(free,free2(1),free2(2))
```

If you wish to erase this special meaning associated with **free** you may do so by supplying no subsequent function names. For example, the following removes the built-in meaning of free.

```
-function(free)
```

Please note that function return values and arguments are still governed by declarations and definitions as they appear in the source code. The special meanings assigned or removed by this operation are those described above and no others. It is in this sense that we can say that realloc is like **free**.

Function0 can have a subscript as well as Function1, etc. In fact it is even more correct to say:

```
-function(free(1),myfree(1))
```

This says that whatever special meaning is associated with the first argument of free should also be associated with the first argument of myfree. Since there are no other arguments of free with special meaning and since free has no special return meaning, omitting the subscripts works fine.

To transfer the return meaning of exit to another function you may use one of:

```
-function(exit,myexit)
```

or

```
-function(exit(0),myexit(0))
```

The zero subscript refers to the return value.

It is common to mix the special return meaning with the -printf meaning. Thus

```
-function(exit,print_and_exit)
-printf(1,print_and_exit)
```

says that `print_and_exit` accepts a format and doesn't return.

CHAPTER 8

Language Extensions

This chapter describes generally-accepted, non-K&R extensions to the C language which have been optionally incorporated into *STATIC*. These features, for the most part, are included in the new ANSI C Standard

8.1 ANSI Extensions

8.1.1 The void Type

`void` is a reserved word (unless the flag **`-fvo`** is set) and is treated in a manner consistent with ANSI C. Functions declared as void are assumed to return no value. Inconsistencies in this regard, obtained from either return statements or calls, are flagged. A pointer to **`void`** is considered a universal pointer, i.e., one that can be assigned or compared freely to any data pointer without an error report.

Finally, to invoke a function that returns a value in order to obtain only its side effects, one may precede that function with a void cast as in:

```
(void) f();
```

Helpful Hint: If your compiler does not support the void type you might consider a definition such as:

```
#ifdef _lint
#define CALL (void)  /* quiets STATIC down
*/
#else
#define CALL
#endif
```

which you would use as:

```
CALL f();
```

if you were interested only in a function's side-effects.

8.1.2 Function Prototypes

Function declarations may optionally contain, within parentheses, type information that indicates the number and the expected types of the argu-

ments. Such a parenthesized construct is called a prototype. Dummy names of parameters may be included for clarity. An ellipsis indicates that an indefinite number of other arguments of arbitrary types may follow the last argument.

For example:

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

void printf( char * format, ... );
```

designate respectively that `strcpy()` is to be called with two character pointers as arguments (and is to return a character pointer) and `printf()` is to be called with at least one argument (a character pointer) and this may or may not be followed by additional arguments. To give an explicit indication that no arguments are allowed, the void keyword is used. For example:

```
int status(void);
```

indicates that `status()` expects no arguments. Had void been omitted, no parameter type information would have been inferred. Function prototypes may also be used for function definitions. Thus:

```
double sum (double x, double y)
   {
   return   x + y;
   }
```

is a valid function definition. (See Section 7.6 - “Prototype Generation” on page 172.)

8.1.3 Enumerations

The enum data type is supported by *STATIC*. In this description, we assume a basic familiarity with this facility. An enumerated data type must first be declared. For example

```
enum primary { red = 1, yellow, blue };
```

declares an enumerated type named primary. Then, the type is used to define enumerated data objects as in: .

```
enum primary x, *px;
```

Finally, these data objects can be used. Use is generally limited to assignment, argument passing and testing for equality.

Enumerated types are processed at one of three levels, strict, loose and intermediate. In the strict model, enumerated type values may only be assigned to variables, passed to parameters, or compared with values of the same enumerated type. At the loose level, which is the model

employed by the ANSI C Standard, an enumerated type value is regarded semantically as an integer. It may be employed in any context that expects an integer and enumerated type variables may be assigned any integral value. This may be done at the strict level only through the use of casts. An intermediate level is to allow use of `enum` 's as integers but to disallow assignment of integers into enum 's. For example:

```
enum food { pear, bread, milk } food1,
food2;

food1 = pear;
food1 = 25;
food2 = food1 + 1;
food2 = (enum food) ( (int) food1 + 1 );
food2 = (enum food) (food1 + 1);
```

In the loose interpretation (the integer model), all five of the assignment statements are correct. In the strict and intermediate models, the second and third are flagged. The fourth represents the modifications made to the third to make it adhere to the strict model. The fifth is a modification to the third to make it adhere to the intermediate model. The default model is the strict model. The intermediate model is obtained by inhibiting Warning 641 (-e641). The loose model is obtained by enabling flag fie (integer model for enumerations).

8.1.4 signed

If character data is by default unsigned (see the flag **fcu**) then, to obtain a signed byte you need to use the signed reserved word as in

```
signed char x;
```

8.1.5 const and volatile

The identifiers `const` and `volatile` are reserved words. const identifies data (possibly through indirection) as not modifiable. A judicious use of `const` can provide important clues to *STATIC* as to how data is being used. (See Section 7.4 - “const Checking” on page 170.) (See Section 7.5 - “volatile Checking” on page 171.)

8.1.6 Trigraphs

For systems that do not have the full ASCII character set, the ANSI standard defines the following correspondence:

Trigraph	Char	Trigraph	Char
??=	#	??<	{
??(	[	??)	]
??’	^	??>	}
??!	|	??-	(tilde)

TABLE 1 Translations for ANSI standards

These translations are supported in both string and character constants as well as source code. A message (739) is issued if the sequence occurs within a constant.

In addition to the `L` (or `l`) suffix, the `U` or `u` suffix is supported to identify a constant as unsigned. For example 50000u is typed `unsigned`.

8.2 Non-ANSI Extensions

8.2.1 // Comments

The sequence / / introduces a comment that extends up to and not beyond the end of the line. For example:

```
n = 0;  // zero n
```

This construct is permitted by a number of compilers but is not strictly ANSI and may be disabled using the -A option.

8.2.2 Memory Models

Memory models have been introduced into a number of C compilers to support the Intel 8086 through 80286 chips and, in some cases, the 80386 and 80486 chip. If you are not concerned with the segmented architecture of these chips you can ignore this section.

There are four distinct memory models and these can be selected by one of the -m ... options described in that section (See Section 3.7.7 - “Strong Typing Options” on page 61.).

option	model name	data pointers	program pointers
default	small	near	near
-mD	large data (compact)	far	near
-mP	large program (medium)	near	far
-mL	large	far	far

TABLE 2 Memory Models

In addition to selecting a memory model, it is possible for a programmer to override the default for any particular pointer. For example:

```
char far *p;
```

indicates that p is a pointer to a far char and is hence a far pointer. In a similar way, pointers can be declared to be near and huge (huge is taken as a synonym for far). Data objects and functions can also be declared as having the property of near or far and pointers to such objects automatically become near or far as appropriate.

It suffices to say that *STATIC* supports the Microsoft conventions for the use of these keywords and that these can be enabled (if they are not pre-enabled in your implementation) by selecting the option +rw(*ms). This requests all the Microsoft reserved words. You may prefer to turn on just one or two of these reserved words. For example: `+rw(near,far)` enables just near and far.

It is also possible to disable these reserved words. by using the option -rw(*ms). The -A option serves to flag such constructs.

Your version of *STATIC* is configured to have the system default sizes for near and far pointers to program and data. For cross-staticing these can be set explicitly using a variation of the **`-sp ...`**.

`-sp`*`N #`* Indicates that the size of **`near`** pointers (both of program and data) is # bytes.

`-sp`*`F #`* Indicates that the size of **`far`** pointers (both program and data) is # bytes.

`-sp`*`FD #`* Indicates the size of a **`far`** data pointer.

`-sp`*`FP #`* indicates the size of a **`far`** program pointer.

`-sp`*`ND #`* Indicates the size of a **`near`** data pointer.

`-sp`*`NP #`* Indicates the size of a **`near`** program pointer.

8.3 Additional Reserved Words

Because of their wide-spread use under MS-DOS, the Microsoft keywords: **near**, **far**, **huge**, **pascal**, **fortran**, **coddle**, and **interrupt** (as well as these same keywords preceded by '_') are supported by default. The meanings of these keywords reflect those of the Microsoft C compiler.

CHAPTER 9

Preprocessor

This chapter discusses STATIC ANSI and non-ANSI as well as include processing.

9.1 Preprocessor Symbols

NOTE: _lint is used so that STATIC is compatible with the standard Lint program.

_lint The special preprocessor symbol _lint is pre-defined in case it is necessary to determine whether STATIC is processing the file.

For example, if you have a section of code that is unacceptable to STATIC for some reason, you can use _lint to make sure that STATIC doesn't see it. Thus,

```
#ifndef _lint
...
Unacceptable coding sequence
...
#endif
```

will cause STATIC to skip over the elided material. The following pre-defined identifiers begin and end with double underscore and are ANSI compatible.

```
__TIME__ The current time
__DATE__ The current date
__FILE__ The current file
__LINE__ The current line number
__STDC__ Defined to be 1 if pure ANSI-com-
patibility is needed ( -A ); else is
defined to be 0. See Section 10.7.7.
```

9.2 include Processing

1. When a #include “filename “ directive is encountered There is first an attempt to fopen the named file. If the fdi flag is OFF the name between quotes is used. If the fdi flag is ON, the directory of the including file 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. There is an attempt to fopen the file by the name provided. If the include directive is of the form #include filename then the processing is the same except that step 1 is bypassed.

9.3 ANSI Preprocessor Facilities

The preprocessor facilities described in this section follow the ANSI C Standard. They are automatically available within STATIC. However, if the K&R preprocessor flag is set (fkp) their use will be flagged.

9.3.1 Initial White Space

Preprocessor directives (those beginning with #) may optionally be preceded with blanks and / or tabs.

9.3.2 #elif expression

The #elif (else if) directive can be used within a #if ... #endif to avoid multiple #if levels. Any number of #elif 's may be used at the same level followed optionally by a #else.

For example:

```
#if x
   text 1
#elif y
   text 2
#elif z
   text 3
#else
   text 4
#endif
```

can be used in place of a much more complex sequence ending in three #endif 's.

defined(name)

The expression defined(name), when used in a #if statement (or #elif statement), is considered true (=1) if the name had been previously defined; otherwise it is considered false (=0). Thus:

```
#ifdef alpha
```

is equivalent to

```
#if defined(alpha)
```

However, the defined construct is considerably more flexible. Consider:

```
#if defined(alpha) || n
   text 1
#elif defined(beta)
   text 2
#endif
```

This may be done with purely K&R constructs but at a considerable loss in clarity and brevity.

9.3.3 #include name

If name is some pre-defined name whose value is a quoted (or <t> bracketed) filename then the named file is included. Thus:

```
#define alpha "abc.h"
#include alpha
```

causes file abc.h to be included.

9.3.4 #pragma

is a construct that allows users to pass compiler-dependent information to particular compilers without complaint from other compilers. STATIC simply doesn't complain.

9.3.5 #error

This construct is used to halt compilation and to print the information following error on the line. If you set the continue-on-error flag (fce) processing will continue.

9.3.6

A single # on a line by itself is a no-op.

9.3.7 ## Pasting operator

The ANSI ## Pasting operator is supported.

```
#define variable(n) var ## n
```

will, for variable(1), return var1. This means that macros can "manufacture" identifiers. The only way to do this earlier was via a construct of the form: VAR()n where VAR() would return var. The Unix-style pasting procedure (employing a comment to perform pastes) is also supported. For example:

```
#define variable(n) var/* */n
```

works the same as above.

9.3.8 # Stringize operator

The # Stringize operator is supported. For example:

```
#define display(var) printf( #var " =
%d\n" , var );
```

serves to display a variable. Both the name and the value of the variable passed to the macro are printed. The construct #var produces "var"; note that successive string constants are treated as a single string constant.

This is ANSI and is needed to make the # stringize operator effective.

9.4 Non-ANSI Preprocessing

9.4.1 #assert

#assert is supported to conform with Unix V Release 4. Thus

```
#assert predicate ( token-sequence )
```

will assume the truth of the predicate when tested against the indicated token-sequence in a preprocessor conditional. Without the parenthetical expression, predicate is established to exist. For example,

```
#assert machine(pdp11)
```

makes

```
#if #machine(pdp11)
```

true.

A #unassert preprocessor directive with the same syntax as #assert undoes the effects of #assert and is compatible with Unix. See also option -a # ... .. (See Section 3.7.6 - "Compiler Customization Options" on page 58.)

9.5 User-Defined Keywords

STATIC might stumble over strange preprocessor commands that your compiler happens to support. For example, some Unix system compilers support #ident.

Since this is something that canNOT be handled by a suitable #define of some identifier we have added the +ppw(command-name) option (Pre-Processor Word). For example, +ppw(ident) will add the preprocessor command alluded to above, recognizes and ignores the construct. (See Section 3.7.7 - "Strong Typing Options" on page 61.).

CHAPTER 10

Additional Notes

This chapter discusses how the size of scalars may affect your report results.

10.1 Size of Scalars

Since the user of *STATIC* has the ability to set the sizes of various data objects. See the size options in the section that describes them (See Section 3.7.4 - “Size Options” on page 51.), the reader may wonder what the effect would be of using various sizes.

Several of the loss of precision messages (712, 734, 735 and 736) depend on a knowledge of scalar sizes. The options **`-ean`** and **`-epn`** only suppresses long / int / short mismatches if they are the same size. Similarly, options **`-eas`** and **`-eps`** depend on the sizes of data items. The legitimacy of bit field sizes depends on the size of an **`int`**. Warnings of format irregularities are based in part on the sizes of the items passed as arguments.

One of the more important effects of type sizes is the determination of the type of the result. The types of integral constants depend upon the size of int s and long s in ways that may not be obvious. For example, even where **`int`** s are represented in 16 bits the quantity:

```
35000
```

is long and hence occupies 4 (8-bit) bytes whereas if int s are 32 bits the quantity is a four byte **`int`**. If you want it to be **`unsigned`** use the **`u`** suffix as in **`35000u`** or use a cast.

Here are the rules: (these ANSI rules may be partially suppressed with the **`fis`** flag) the type of a decimal constant is the first type in the list (**`int, long, unsigned long`**) that can represent the value. The maximum values for these types are taken to be:

Largest	is 1 less than 2 raised to the power:
`int`	**`sizeof(int)*`** *bits-per-byte* - 1
`unsigned`	**`sizeof(int)*`** *bits-per-byte*

long **sizeof(long)*** *bits-per-byte* - 1

unsigned long **sizeof(long)*** *bits-per-byte*

The quantities **sizeof(int)** and **sizeof(long)** are based on the **-si #** and **-sl #** options respectively. The type of a hex or octal constant, however, is the first type on the list (**int, unsigned int, long, unsigned long**). For any constant (decimal, hex or octal) if it has a **u** suffix, one selects from the list (**unsigned int, unsigned long**). If an **L** suffix, the list is (**long, unsigned long**). If both suffixes are used then the type must be **unsigned long**.

The size of scalars enters into the typing of intermediate expressions in a computation. Following ANSI, STATIC uses the so-called *value-preserving* rule for promoting types. Types are promoted when a binary operator is presented with two unlike types and in passing function arguments. For example, if an int is required in an operation and if an unsigned short is presented, then this is converted to **int** provided that an **int** can hold all values of an **unsigned short**; otherwise, it is converted to **unsigned int**. Thus the signedness of an expression can depend on the size of the basic data objects.

10.2 !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 -e506 */ (!0) \\
/*lint -restore */
```

Note: The use of the **()** 's around **!0** are needed to force parsing of **!0** to end before the ' **-restore** '.

CHAPTER 11

Common Problems and Applications

This chapter is split into two main parts. The first part describes how to handle common problems and the second part describes how to use *STATIC* in a practical manner.

11.1 Common Problems

11.1.1 Too Many Messages

It should be emphasized that suppressing a message does not alter the behavior of *STATIC* 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.

To set a warning level, use option **`-w`**.

11.1.2 Warning 516

A surprising diagnostic (surprising to at least some programmers) is issued for the following:

```
int f(char);
...
int f(c) char c; { ...
```

This results in **`Warning 516 f has argument type conflict with ...`**. This is an example of mixing a new-style function prototype (the first declaration above) with an old-style function definition (the second declaration above). With an old-style function definition, the compiler adjusts the parameter's type from **`char`** to **`int`**. See, for example, K&R, 1st edition, page 205; K&R, 2nd edition, page 202, or Harbison & Steele, 1st edition, page 231 or 2nd edition page 228. The ANSI standard addresses this issue in another chapter (See CHAPTER 8 - Language Extensions” on page 185.). A prototype, on the other hand, has no implicit promotion associated with it.

There are several ways around the problem. Since old-style function definitions are now deprecated by ANSI C, you could use the new-style definition:

```
int f(char c) { ...
```

If you're worried about portability to not-yet-ANSI compilers (in which case only *STATIC* should be looking at the prototype), you can change the prototype to:

```
int f(int c);
```

If you're worried about the type discrepancy you can use a special type for this purpose, say XCHAR, for eXtended char. You would then have

```
typedef int XCHAR;
int f(XCHAR);
int f(c) XCHAR c; { ...
```

There are also two additional means available to cope with this problem.

`-eai` suppress complaints about sub-Integer type mismatches. (See Section 3.7.1 - “Error Messages Options” on page 32.)

Also, flags **`fxc`** and **`fxs`** can be used to turn char and short parameter declarations into exciting new type-checks. For example, the **`fxc`** flag (eXact Character flag) takes the **`char`** declaration within the old-style function definition literally. Then all arguments passed to function f() must resemble the exact un-promoted type of the argument. See the section that describes exact parameter matching information (See Section 7.7 - “Exact Parameter Matching” on page 174.).

11.1.3 Error 123 Using Min or Max

Some Microsoft C users have been confused about getting error 123 when all they do is have a declaration of the form:

```
int min;    OR    int max;
```

Actually, somewhere in the module is an include of **“stdlib.h”** which defines macros **`min()`** and **`max()`**. If you do not want *STATIC* to complain about this dual use (because they're used all over the place), simply suppress the message with **`-e123`** or **`-esym(123,min,max)`**. See the section that gives further information on using error message options (See Section 3.7.1 - “Error Messages Options” on page 32.).

11.1.4 LONG_MIN Macro

STATIC will occasionally issue a warning (501 and / or 569) when using the **`LONG_MIN`** macro from your compiler's **`limits.h`** header file. We

have found the following variations in the definition of **LONG_MIN** among several different compiler vendors.

```
#define LONG_MIN -2147483647         /* OK
*/
#define LONG_MIN 0x80000000L          /*
Warning */
#define LONG_MIN (-2147483647-1)     /* OK
*/
#define LONG_MIN ((long)0x80000000L) /* OK
*/
#define LONG_MIN -2147483648L         /*
Warning */
#define LONG_MIN (-(2147483647L)-1)  /* OK
*/
#define LONG_MIN -2147483648          /*
Warning */
```

For those that we issue a warning, the quantity is typed **unsigned long** and if you used this type as in:

```
if( n > LONG_MIN ) ...
```

you would find that the test which should almost always succeed would almost never succeed. Perhaps you should alert your compiler vendor.

11.1.5 Plain Vanilla Functions

By a plain vanilla function (or canonical function) we mean a function declared without a prototype. For example:

```
void f();
```

Not too many programmers realize that such a function is incompatible with one that is prototyped with a **char**, **short**, or **float** parameter or has an ellipsis. We warn you (type difference = '**promotion**' or '**ellipsis**') but the warning can cause confusion if you do not realize the difference.

When a call is made to such a function the compiler must decide which, if any promotions to apply to the arguments. Since the declaration said nothing about arguments, a standard (i.e., canonical) set of promotions is applied. According to ANSI, char's and short's are promoted to int, and float's are promoted to double. Also the argument list is presumed fixed so that registers may be used to pass arguments.

Prototypes can inhibit such promotions; if **f** was declared:

```
void f( char, short, float );
```

All three promotions would be inhibited. For this reason this declaration is incompatible with the earlier declaration and you receive a warning. If **f** was 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 standard.

11.1.6 Strange Compilers

You may want to run *STATIC* on programs that have been prepared for compilers that accept strange and unusual constructs, and for which there is no custom compiler options files. There are a number of options you can use to get *STATIC* to ignore such constructs. Chief among these are the **-d**, **+rw** and **+ppw** (See Section 3.7.7 - “Strong Typing Options” on page 61.). But also check the section that describes customization Facilities (See Section 3.7.5 - “Compiler Vendor Options” on page 54.) for additional options to help cope with the truly extraordinary.

11.2 Real-Life Applications

The comments in this section 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 *STATIC* against a large C program that has not previously been run, 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 *STATIC*.

Stories of its effectiveness, however, are legendary. *STATIC* was, of course, passed through itself and a number of subtle errors were revealed in spite of exhaustive prior testing. I tested a public domain grep that I never dared use because it would mysteriously bomb. *STATIC* found the problem-- an un-initialized pointer.

It is not only necessary to test a program once but it should be continuously tested throughout a development / maintenance effort. Early in *STATIC*'s development, we spent a considerable effort, over several days, trying to track down a bug that *STATIC* would have detected easily. We learned our lesson and were never again tempted to debug code before running *STATIC* on it.

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 700 level messages (informational messages) by the option **-e7??** (See Section 3.7.1 - "Error Messages Options" on page 32.)or **-w2** (See Section 3.7.7 - "Strong Typing Options" on page 61.). Then work to correct only the errors associated with the messages that remain.

The policy can be automatically imposed by incorporating the error suppression options in a batch file and/or **.lnt** file (examples shown next) and it can gradually be strengthened as time or experience dictate.

Experience has shown that running *STATIC* 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.

11.2.1 An Example of a Policy

An example of a set of practices with which I myself can comfortably live is as follows.

I make frequent use of C's ability to test the result of an assignment by using a construct such as:

```
if( a = value )
{ ...
```

So I routinely suppress error message 720 by the option:

```
-e720
```

Someday, if I have the time and if I become convinced that I won't lose efficiency, I might convert all these to:

```
if( (a = value) != 0 )
{ ...
```

but for now I'll take my chances.

But note that recent converts from the Pascal community should perhaps choose the latter construct and not inhibit 720.

At one time I would have suppressed pointer-pointer messages with the **-epp** option. This was to avoid the need for excessive casting. For example the statement:

```
p = malloc(n);
```

would usually require a cast since **malloc()** would normally be declared as returning a pointer to an object of different type than what **p** was pointing to. Similar remarks could be made regarding:

```
free(p);
```

Excessive casting is not a good idea because otherwise suspicious constructs are not reported on. With the introduction of **void *** much of the casting can be avoided. **malloc()** should be declared as returning a **void *** and **free()** should be declared as accepting **void ***. Hence I no longer use **-epp**.

As an example of a particular coding style, I frequently mix unsigned and signed quantities. Hence, I use the message suppression options:

```
-e502  -e713  -e737  -eau
```

(502 involves applying ~ to a **signed** quantity, 713 involves assigning **unsigned** to **signed**, 737 is loss of sign, and **eau** suppresses messages based on the fact that an argument and a parameter disagree in that one is **signed** and the other is **unsigned**). Some **signed/unsigned** messages are left ON, such as Warnings 568 comparing unsigned in certain ways to zero and 573, and 574 (mixing s**igned/unsigned** in certain operators).

A message suppressed with some sense of guilt is **734** (sub-integer loss of precision). This message can catch all sorts of things such as assigning **ints** to shorts when **int** is larger than short, assigning oversize ints into chars, assigning too large quantities into bit fields, etc. However, in too many instances one is assigning an int to a char in the normal course of coding. C encourages chars to be kept in ints because **EOF** is **-1**. However, if you want the additional checking inherent in not suppressing message **734** then do the following. After a character is read into an **int** and checked to be not **EOF** immediately assign the value to a **char** via a cast.

I suppress messages about shifting **ints** (and **longs**) left but I want to be notified when they are shifted right as this can be machine-dependent and is generally regarded as a useless and hazardous activity. Therefore, I use **-e701** **-e703**.

I routinely employ functions without a prior declaration allowing them to default to int. Therefore I use option **-e718** (function not declared).

I tend not to call in the presence of a prototype. Calling with a prototype in scope is not yet completely portable and can cause quiet unintended conversions. Hence I routinely use option **-e746** to suppress the “not called with a prototype” message. I place my list of favorite error-suppression options in a file called options.lnt. It looks like this:

```
-e720// test of assignment
```

```
-e502 -e713 -e737 -eau // unsigned-signed
-e734// sub-integer loss of info
-e701 -e703// shifting int left is OK
-e718// undeclared function
-e746// allow calls w/o prototypes
```

11.2.2 The Setup

I will place my reference to options.lnt along with my compiler options file within an indirect file called std.lnt. std.lnt looks, in its simplest form, like this:

```
// Standard lint options
co-xxx   options.lnt
```

This **std.lnt** is placed in a globally accessible directory. I then refer to **std.lnt** from within a command script file. The advantage of doing the double indirection is that for special projects I will use a special **std.lnt** that may include options in addition to those within **options.lnt**. The specialized **std.lnt** is placed within the project directory. All this is to be done using the same basic command script file.

11.2.3 Using Lint Object Modules

For large projects (more than several source modules), I use Lint Object Modules (See CHAPTER 6 - Lint Object Modules" on page 161.). My make file set up is similar to that described in Section 6.4. A typical make file has the form:

```
c.lob:
lint -u make.lnt $* -oo

  m1.lob: ml.c

  m2.lob: m2.c

  m3.lob: m3.c

  m4.lob: m4.c

  m5.lob: m5.c

  project.lob: m1.lob m2.lob 0. . .7
m5.lob
lint make.lnt *.lob
```

In the above, make.lnt contains those things that the batch file lin.bat (described earlier) provided. In particular it contains:

```
-ic:\\lint\\
std.lnt
-os(temp)
+v
```

11.2.4 Summarizing

In summary, establish procedures whereby *STATIC* may be conveniently accessed for a variety of purposes. Use *STATIC* on 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. Use *STATIC* at full strength on new projects.

References

1. Kernighan, B. and D. Ritchie;
   The C Programming Language,
   Prentice Hall, 1978 (First Edition),Englewood Cliffs,NJ;
   1988 (Second Edition).
2. *ANSI Standard X3.159-1989*
   American National Standards Institute,
   New York, 1989.
3. Harbison, S.P. and G.L. Steele, Jr.;
   A C Reference Manual,
   Prentice Hall, Englewood Cliffs NJ;
   1984 (First Edition), 1988 (Second Edition).
4. Plum, Thomas;
   Notes on the Draft C Standard,
   Plum Hall Inc.,
   Cardiff NJ 08232.
5. Ward, Robert;
   Debugging C,
   Que Corporation, Indianapolis IN, 1986.
6. Jaeschke, Rex;
   Portability and the C Language,
   Hayden Books, Indianapolis IN, 1989.

Index

Symbols

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

X