Compiler Guide

SYMANTEC.

C++

DEVELOPMENT SYSTEM FOR
POWER MACINTOSH

VERSION 8

POWER MACINTOSH/
MACINTOSH

Symantec C++ for Power Macintosh

Compiler Guide

Credits

Documentation John Minniti, Jeanne Munson, Stephen Raphel, and Susan Rona

Development David Bustin, Thomas Cardozo, Thomas Emerson, Bob Foster, Udi Kalekin, Paul Kaplan, Doug Knowles, Jim Laskey, John Micco, Pat Nelson, Mark Romano, Phil Shapiro, and Rob Vaterlaus

Quality Assurance Celso Barriga, Colen Garoutte-Carson, Constantine Hantzopoulos, Kevin Irlen, Yuen Li, and Christopher Prinos

Technical Support Glenn Austin, Mark Baldwin, Craig Conner, Colen Garoutte-Carson, Rick Hartmann, Michael Hopkins, Steve Howard, Scott Morison, and Kevin Quah

Project Management Constantine Hantzopoulos, Doug Knowles, and David Neal

Product Management David Allcott

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL SYMANTEC'S LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY SYMANTEC'S LICENSOR) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF SYMANTEC'S LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

SYMANTEC'S Licensor's liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to $50.

Contents

Welcome 1

Welcome to Symantec C++ for Power Macintosh. This manual contains the reference information for the compilers included with Symantec C++ for Power Macintosh. The Symantec C++ package includes the Symantec C and C++ compilers and libraries as well as the entire Symantec development environment.

Contents

If You Are New to the Symantec Environment

The *Symantec C++ User's Guide and Reference* describes how to use the powerful Symantec development environment. It is a procedure-oriented book that takes you step-by-step through the process of developing an application on the Power Macintosh.

The *User's Guide and Reference* also contains tutorials that can help you learn how to use Symantec C++ for Power Macintosh. Working through the tutorials is a good way to get started with Symantec C++.

If You Are Learning C or C++

The documentation included with Symantec C++ for Power Macintosh does not attempt to teach C or C++. Rather it explains the Symantec development environment, its compilers, and other associated tools.

If you are relatively new to C or C++, you might find the following suggestions useful in gaining proficiency with Symantec C++ for Power Macintosh. Read the tutorials in the *Symantec C++ User's Guide and Reference* to learn how to run simple programs using Symantec C and C++.

If you're learning C or C++ from a book written for UNIX or MS-DOS computers, you'll want to use *THINK Reference* (in the Online Documentation folder) to look at the:

- "Standard Libraries Intro" in the Standard Libraries Reference database
- "Console Package Intro" in the Standard Libraries Reference database
- "Introduction to Streams" in the IOStreams Reference database

What Is Symantec C++ for Power Macintosh

Symantec C++ for Power Macintosh is a unique development environment for the Power Macintosh. It features very fast C and C++ compilers, powerful optimizers, a resource compiler, a fast linker, an integrated debugger, a full-featured text editor, an auto-make facility, an object-oriented GUI builder, and a project organizer that holds the pieces together. Because the editor, the compilers, and the linker are components of the same application, Symantec C++

knows when edited source files need to be recompiled. If you edit a header file, the auto-make facility recompiles the source files that depend on it for declarations.

With Symantec C++, you can build Power Macintosh applications. The standard C libraries include the functions specified in the ANSI C standard, as well as some additional UNIX operating system functions. The C++ libraries include IOStreams, a flexible extensible class library for doing input and output, Complex, a library that lets you do mathematical operations with complex numbers, and a version of STL, the C++ standard template library that contains many classes usable in a variety of contexts.

Symantec C++ for Power Macintosh also contains separate tools to create 68K applications and code resources. Refer to the online documentation for additional information on these tools.

You can run your program from the Symantec development environment as you work on it. Your program runs exactly as if you had opened it from the Finder, not under a simulated environment. Your program runs in its own partition while the Symantec Project Manager remains active, so you can examine and edit your source files as you watch your program run.

The Symantec development environment includes a source-level debugger that lets you debug your code exactly as you wrote it; there's no need to translate assembly language back into source code. The debugger lets you set breakpoints, step through your code, debug objects, examine variables, and change their values while your program is running. And because the debugger works together with the development environment, you can edit your source files while you're debugging.

What you need

Symantec C++ for Power Macintosh requires a Power Macintosh and at least 16 megabytes (16MB) of RAM. Large projects require more memory.

The complete Symantec C++ system including Apple Tools, the full 68K native environment, and online documentation takes up about 110MB on your disk, not including your own files. The actual size of your system can be considerably smaller depending on the kinds of

programs you develop. You can customize your installation to use less disk space. Refer to the *Symantec C++ User's Guide and Reference* for information on custom install options.

What Your Package Contains

Your Symantec C++ for Power Macintosh package consists of a CD-ROM, this manual, and the *Symantec C++ User's Guide and Reference.*

What's in This Manual

The chapters in this manual are: "Using the Symantec Compilers," "Calling Toolbox Routines," "Compiler Reference," "Compiler Options Reference," "Porting Code," "Using the Standard Libraries," "Using Symantec Rez," and the appendixes "Language Reference" and "Error Messages." Each chapter begins with an introduction that describes what's in the chapter.

Welcome	This is the section you're reading. It describes the *Compiler Guide.*
Using the Symantec Compilers	How Symantec C++ for Power Macintosh compiles your source files. It also tells you how to change language settings, choose C or C++ compilers, and use the global optimizer.
Calling Toolbox Routines	How to call the Macintosh Toolbox routines from your programs. It also explains how to write Pascal callback routines.
Compiler Reference	Aspects of the Symantec C and C++ implementations that are not part of the C and C++ language definitions.
Compiler Options Reference	Special features and extensions in the Symantec C and C++ implementations. It discusses special object types, pragmas, and Macintosh-specific extensions to the C and C++ languages.

Porting Code	How to port code from 68K and MPW C/C++ to Symantec C/C++. It also contains hints on how to port from other C/C++ implementations.
Using the Standard Libraries	The standard libraries that come with Symantec C++ for Power Macintosh, and how to use them with your project. The chapter also explains how to modify them for your own purposes, how to use the standard libraries supplied by Apple, and how to use THINK Reference.
Using Symantec Rez	How to write text files that compile with Symantec Rez to produce resource files.
Language Reference	This appendix describes Symantec's implementation of the C and C++ languages.
Error Messages	This appendix describes all the error messages generated by Symantec C, Symantec C++, and Symantec Rez.

Conventions Used in This Manual

The names of menus, commands, and dialog boxes are in **boldface**.

Names of files, code fragments, resource names, function names, folders, and variables appear in `typewriter face`.

All numbers are decimal. Hexadecimal numbers are written in C notation: `0x3EFA`.

Library and window names appear with the first letter capitalized.

Metanames are *italicized.*

In this manual, the term "Toolbox routine" means any routine described in *Inside Macintosh.*

What You Should Know

This manual assumes you already know, or are learning, how to program in C or C++. If you're just getting started in C or C++, the Symantec development environment is a great platform.

If you're planning to write Macintosh applications, you should be familiar with the Macintosh Toolbox as described in *THINK Reference* or in *Inside Macintosh*, the official reference that describes the many Macintosh Toolbox routines. The Toolbox is the set of operating system and user interface routines that make a Macintosh a Macintosh. *THINK Reference*, included with Symantec C++ for Power Macintosh, is an invaluable tool for learning about the Macintosh Toolbox and the libraries provided by Symantec C++. It's beyond the scope of this manual to show you how the different parts of the Toolbox work together.

Learning C/C++

As the popularity of C and C++ grow, more and more introductory-level books appear on the shelves. Most books assume that you already know how to program in another language. Some books spend time telling you how to use components of a development environment: an editor, a linker, a make facility. If your book discusses a particular development environment, make sure it's tailored to Symantec C++ for Power Macintosh, or at a minimum, the THINK development environment.

Investigating C

Many introductory-level books are available to help you learn C. Some books assume that you're just learning how to program; others assume that you already know how to program in another language.

The standard references for the C programming language are Kernighan & Ritchie's *The C Programming Language, Second Edition* (Prentice Hall) and Harbison & Steele's *C: A Reference Manual* (Prentice Hall). *The C Programming Language, Second Edition* is an update to *The C Programming Language* that incorporates information from the ANSI standard. These books assume that you're already an experienced programmer.

Standard C (Microsoft Press) by P. J. Plauger and Jim Brodie is a guide to writing C programs that conform to the ANSI C standard. Both of the authors were officers of the committee that drafted the ANSI standard.

Software Engineering in C (Springer-Verlag) by Peter Darnell and Philip Margolis is an excellent introduction to the C programming language. This book is ideal for new C programmers who have programmed in other languages.

C Traps and Pitfalls (Addison-Wesley) by Andrew Koenig is a good book for intermediate and advanced C programmers. It contains a detailed discussion of common C programming problems.

Numerical Recipes in C (Cambridge University Press) by Press, Flannery, Teukolsky, and Vetterling is a detailed technical description of numerical methods with implementation examples in C.

Portability and the C Language (Hayden Books) by Rex Jaeschke is a good book on writing portable C programs for advanced C programmers. It gives guidelines on writing programs that can be compiled with different compilers and run on multiple platforms. It points out changes introduced with ANSI C. You'll find it useful if you port code from one platform to another or from an old (pre-ANSI) version of THINK C.

Portable C Software (Prentice Hall) by Mark R. Horton is also about writing portable C programs, but emphasizes porting among C compilers for Unix and MS-DOS.

At the time of this writing, the ANSI standard is described in the *ANSI/ISO 9899—1990, Information Technology-Programming Language C.* The cost is $130, plus 7% for shipping and handling. To order a copy, write or call:

American National Standards Institute (ANSI)
Sales Dept.
11 West 42nd Street
New York, NY 10036
(212) 642-4900

Investigating C++

The standard references for the C++ programming language are *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (Addison-Wesley, 1990) and *The C++ Programming Language, Second Edition* by Bjarne Stroustrup (Addison-Wesley,

1992). These books assume that you're already an experienced programmer. *The C++ Programming Language* includes a ten-chapter tutorial introduction to C++.

When the ANSI/ISO C++ standard becomes a draft, Symantec will compare this implementation to that standard.

Other books for learning C++ include:

Learn C++ on the Macintosh by Dave Mark (Addison-Wesley, 1993) is for beginning C++ programmers. It was written specifically for use with Symantec C++ for 68K, which is similar to the Power Macintosh version.

The C++ Primer, 2nd Edition by Stanley Lippman (Addison-Wesley, 1992) is a solid, easy-to-read introduction to C++. It does not assume knowledge of C, but does assume knowledge of some modern block-structured language.

Object-Oriented Programming in C++ by Ira Pohl (Benjamin/Cummings Publishing, 1993) teaches both C++ and object-oriented programming techniques.

The C++ Answer Book by Tony L. Hansen (Addison-Wesley, 1990) contains useful examples, questions, and answers. Although it was written as a companion book to the first edition of *The C++ Programming Language*, it is still current and informative.

C++ for C Programmers by Ira Pohl (Benjamin/Cummings Publishing, 1989) is for experienced C programmers who want to learn C++. It introduces the C++ features that C programmers can put into immediate practice.

The IOStreams Handbook by Steve Teale (Addison-Wesley, 1993) is a comprehensive, detailed explanation of the standard input and output library used in C++. Teale shows programmers how to use IOStreams, provides reference material for the IOStreams classes, illustrates how to provide input-output facilities for user-defined types, and explains how to extend the IOStreams system. This book will help programmers, both novice or experienced, to expand and manipulate IOStreams, and to make more sophisticated use of facilities in their own programs.

Advanced C++ Programming Styles and Idioms by James O. Coplien (Addison-Wesley, 1993) is for programmers, already knowledgeable in C++, to help develop their expertise. It provides a feel for the styles and idioms of C++.

To stay on the cutting edge of object-oriented technology and C++ programming, you may want to subscribe to the following magazines:

The C++ Report: The International Newsletter for C++ Programmers, JPAM SIGS Publication Group, 310 Madison Ave., Suite 503, New York, New York 10017.

The Journal of Object-Oriented Programming, JPAM SIGS Publication Group, 310 Madison Ave., Suite 503, New York, New York 10017.

Using the Symantec Compilers

2

Different compilers implement C differently, even if they conform to the ANSI standard. Similarly, C++ compilers can differ in the way they implement C++. This chapter explains how to use the unique features of the Symantec C and C++ compilers, including how to compile source files, how to use precompiled headers, and how to set options that affect the way your source files are compiled. In addition, this chapter discusses code optimization, and lightly touches on how Symantec C++ complies with the draft ANSI standard for the C++ language.

Contents

Compiling Source Files

Unlike traditional compilers, Symantec C++ for Power Macintosh doesn't generate separate object files from your source files. Instead, the Symantec C and C++ compilers put all the object code into the project document. Although you can compile files manually, most of the time you use the auto-make facility to compile your files.

Note

When compiling for the external linker (PPCLink), Symantec C++ for Power Macintosh produces object files, which are standard object format for the Power Macintosh, that are *not* placed in the project document. For more information on the external linker, see the *Symantec C++ User's Guide and Reference.*

Choosing a compiler

Prior to creating your source files, you should determine if you will be using C, C++, or a combination of the two compilers. If you are not sure which language to use, you can use the following information as a guide.

Create your source in C if your procedure is highly algorithmic or computationally intensive. Create your source in C++ to take advantage of strong type checking for function arguments and variable assignments. C++ is more conducive to large programming projects where strict guidelines must be maintained amongst multiple programmers.

Even though C++ is mostly a superset of C, there are some C constructs that will not compile under C++. In cases where you have a considerable amount of previous C code that does not compile under C++, use both C and C++.

Compiling files not in the project

You can add a source file to your project and compile it in one step. First, create your source file with the Symantec Editor. Save your file in the same folder as the project document. Make sure that your Symantec C++ file name ends in `.cp` or `.cpp`, and your Symantec C

file name ends in `.c`. By default, the Symantec C++ compiler only compiles files that end in `.cp` or `.cpp`, and the Symantec C compiler only compiles files that end in `.c`.

Note

You can use the Extensions options in the **Project Options** dialog to change the file extensions that the Symantec Project Manager uses to determine which compiler (C, C++, Rez, etc.) to use to compile source files ending in a given extension.

Next, choose **Compile** from the **Build** menu. A dialog shows you how many lines Symantec C or C++ has compiled. If there are no errors in the source file, the project manager adds the file and its object code to the project.

Note

The Symantec Project Manager in Symantec C++ for Power Macintosh uses PowerPC C as the name of the Symantec C compiler and PowerPC C++ as the name of the Symantec C++ compiler.

Compiling files already in the project

If you want to compile a file that is already in the project, click its name in the project window and choose **Compile** from the **Build** menu. Once a file is in the project, you don't need to open it to compile it.

Checking files without compiling

You may want to check that your source file will compile without actually compiling it. First, save the file (using the extension `.c`, `.cp`, or `.cpp`). Next, choose **Check Syntax** from the **Build** menu. The compiler checks the syntax of the contents of the frontmost Editor window without generating code or adding the file to the Project window. If it's in the project, you don't need to open it to check syntax.

Fixing errors in source files

When Symantec C or C++ detects an error in your source file, it opens the Build Errors window. If there is more than one error, the Symantec Project Manager reacts according to the settings you chose in the **Options** dialog. One setting you can choose is to have all

errors listed in the Build Errors window. If you double-click an error in this window, the source file that contains the error opens in an Editor window and the line that contains the error is highlighted.

Error reporting

The Error Reporting radio button cluster in the Debugging page of the PowerPC C or PowerPC C++ **Project Options** dialog lets you choose how the compilers report errors to you. The choices are Stop at first error, Report the first few errors, and Report all errors in a file. See "Debugging" in Chapter 5 for more information on error reporting.

Note

If you are using the auto-make facility, the compiler will honor the Error Reporting setting (described in the previous section) for that file, report the error(s), and then move on to the next file.

Precompiled Headers

Symantec C++ for Power Macintosh lets you precompile header (`#include`) files. Precompiled headers may contain only declarations and preprocessor symbols. Since precompiled headers are in a format Symantec C++ can use readily, they load significantly faster than text header files.

Another benefit of precompiled headers is that they make the size of the debugging information smaller.

"Calling Toolbox Routines" describes which files are included in the PPC MacHeaders and PPC MacHeaders++ files.

Usually, you use the built-in `PPC MacHeaders` or `PPC MacHeaders++` that are included with Symantec C++ for Power Macintosh. Symantec C uses `PPC MacHeaders` and Symantec C++ uses `PPC MacHeaders++`. These two precompiled header files differ because the information maintained by the compiler is different for C and C++. For example, a template declaration can appear in a C++ precompiled header, but not in a C precompiled header.

Customizing the PPC MacHeaders/PPC MacHeaders++ files

You can change the default files or make your own precompiled headers. For example, you might find that programs you write frequently refer to a header file that is not already in

PPC MacHeaders or PPC MacHeaders++; or they might include some files you never use. You can customize these files to suit the programs you write in the following way:

1. Find the file Mac #includes.c in the Mac #includes folder. (The file Mac #includes.cpp merely includes Mac #includes.c.) Duplicate Mac #includes.c and give it a new name with the proper extension. For example, use My #includes.c with the C compiler, and My #includes.cpp with the C++ compiler. Open the duplicate with the Symantec Project Manager.

2. Search for the files you want to add or remove. The #include statements are enclosed in conditional compilation directives. To add a file, change the #if 0 directive to #if 1. To remove a file, change the #if 1 directive to #if 0. Some files can't be used together. For more information, see below.

3. Choose **Precompile As...** from the **Build** menu. Save the file as PPC MacHeaders++. The best place for PPC MacHeaders++ is in the Mac #includes folder, but you can save it anywhere in the system tree.

Note

System tree refers to the folder containing the Symantec Project Manager application and all folders that are not shielded within it. See "Shielded folders" in Chapter 4 for more information.

The auto-make facility marks the files in the current project for recompilation if you change PPC MacHeaders++ or any other include file that it depends on. To let other projects know that PPC MacHeaders++ has changed, use **Check Dependencies...** in the **Build** menu, and click OK when the dialog appears.

Creating your own precompiled header

If you want to use your own precompiled header, follow these steps:

1. Create a file containing the desired series of #include statements and symbol definitions.

2. Verify that the current project's compiler settings are the ones you want to use to build your precompiled header.

Note

A prefix isn't used when precompiling.

3. Choose **Precompile As...** from the **Build** menu. Name the file and click Save.

You use a precompiled header the same way you use any other header file. Use the `#include` statement to load your precompiled header into your source file. The `#include` statement must be the first noncomment line of your source file.

You can use only one precompiled header per source file. If you `#include <PPC MacHeaders++>` in the project prefix, you can't explicitly include any other precompiled header.

If you don't `#include` any precompiled headers in the project prefix, you can use several different precompiled headers for different source files in your program. You can still explicitly include `PPC MacHeaders++` if you want to use it in particular files.

Note

You can use only one precompiled header per source file.

You can use your custom-precompiled headers for your own files. You can `#include` them explicitly in your source files as long as you don't `#include <PPC MacHeaders++>` or `<PPC MacHeaders>` in your prefix. Judicious use of precompiled headers can significantly reduce compilation time and debugger table size.

Symantec C++ Reports

Symantec C++ lets you look at your source code and finished applications in three different ways: you can see the preprocessor output of a source file, the assembly code a source file produces, and a link map of a finished application.

Viewing the preprocessor output

If you think you have a bug in one of your macros, use the **Preprocess** command in the **Build** menu. It runs the code in the frontmost window through the Symantec preprocessor and displays the result in a new window. The preprocessor expands your macros and includes the contents of your `#include` files. It evaluates your `#if` or `#ifdef` statements. You can save and print the contents of this window as you would any other file.

Note

The preprocessing directives that control conditional compilation are included in the output. This allows you to debug file-inclusion errors as well as macros.

Source in precompiled headers is not included inline in the preprocessed output.

Disassembling your code

Looking at the assembly code that the compiler produces helps you debug your code and assess its efficiency. The **Disassemble** command in the **Build** menu disassembles the code in the frontmost window and displays the result in a new window. You can save and print the contents of this window as you would any other file.

Note

You can only disassemble code that you can compile.

Generating a link map

The Symantec Project Manager's Internal Linker can generate a link map of your build results. The link map lists information about the content of your built application's or shared library's PEF (Preferred Executable Format) container, including import information and a layout of each of the PEF container's three executable sections.

For more information about PEF, refer to the Inside Macintosh PowerPC System Software Manual, *(Addison-Wesley).*

Note

Since an application file or shared library file can contain one or more PEF containers, this section uses *container* to mean a single build unit, either application or shared library.

To generate a link map, turn on the Generate a link map option in the Linker's **Project Options** dialog. The Internal Linker creates the link map only when you use **Build Application** or **Build Library** from the **Build** menu. The name of the map text file is the application or fragment name with `.map` appended. For example, the link map for the `Test` application is `Test.map`. The Symantec Project Manager places the link map in the same folder as the application and overwrites any pre-existing link map.

The first portion of the link map contains general information about the container including the name of the container, the time and date the container was built, and each of the shared libraries required to execute the container. In the following example, the application's name is `test`, it was built on `November 7, 1994` at `9:22 AM` and requires the presence of the `InterfaceLib` and `MathLib` shared libraries in order to execute:

```
File: Test
Date: Monday, November 7, 1994 9:22 AM

Import: InterfaceLib
Import: MathLib
```

Note

`InterfaceLib` and `MathLib` are automatically provided by the Macintosh operating system in ROM.

The next three portions of the link map contain layout information about the container's three executable sections: code, data, and uninitialized data.

The format of each section is generally the same, a section header followed by a table of section content. Each entry in the content table contains an entry point's base address, a TOC offset, a routine descriptor address, a routine glue address, and the entry point's unmangled symbol name. The table is sorted by base address. All values are in hexadecimal. Data sections do not have the descriptor or glue columns. The link map is designed so that it can be read by analysis tools.

For debugging purposes, the base addresses of each section of the link map are set so that an address in the PEF can be mentally mapped to its correct section. Addresses in the code section are in the form 0XXXXXXX, addresses in the data section are in the form 2XXXXXXX, and addresses in uninitialized data are in the form 4XXXXXXX.

Note

TOC offsets are not addresses but offsets from TOC. To compute a TOC address, add the TOC offset to the TOC base address (in the data section).

The code section contains executable code, primarily from the compiled routines generated by the Symantec Project Manager's translators. The code section also contains (although hidden in the link map) glue generated by the Internal Linker that is used by the PowerPC runtime to handle cross-fragment calls. For example:

```
Section: '.text'
Address  TOC       Desc      Glue      Symbol

         00000004            00000000 NewPtr
         00000008            00000018 NewPtrClear
         0000000C            00000030 GetPtrSize
         00000010            00000048 SetPtrSize
...
00000F60 00000028 20000760 00000118 malloc
00001010 0000002C 20000768          calloc
00001120 00000030 20000770          realloc
00001404 00000034 20000778 000001C0 free
00001464 00000038 20000780          alloc
000016A4 0000003C 20000788         malloc_cleanup
...
```

In this example, the first column is blank for the first few entries because they are imported from another container. The C function malloc is at base address F60, calloc is at 1010, and so on.

A routine descriptor is an 8- or 12-byte structure that contains the routine's base address and the address of the TOC that should be used with that routine. A routine descriptor may be blank if the address of the function is not taken and it is not exported from the fragment, or the routine is imported.

For TOC-based calls, the container uses TOC offset 4 to get the address of the routine descriptor for NewPtr and TOC offset 28 for malloc. The routine descriptor for malloc is at address 20000760 in the data section.

The routine glue for malloc is at 118. This glue routine is used to call malloc with the appropriate TOC. Glue may be blank if the address of the function is not taken.

The data section contains initialized global data values such as initialized global variables, initialized static member variables, C++ vtables, strings, floating-point constants, and routine descriptors. The data section also contains the content of the runtime TOC. For example:

```
Section: '.data'
Address  TOC       Symbol

20000000           TOC
...
20001270 00000490  ftype
20001274 00000494  fcreator
...
```

In this example the first entry is TOC (don't rely on the TOC being the first entry in the data section because this may change). The base address of ftype is 20001270 or 1270 bytes from the beginning of the data section. The TOC entry at offset 494 contains the address of fcreator. For obvious reasons, TOC doesn't need a TOC offset.

The uninitialized data section contains global data values such as uninitialized global variables and uninitialized static member variables. All entries in the uninitialized data section are preset to zero by the PEF fragment loader. For example:

```
Section: '.bss'
Address  TOC       Symbol

40000000 00000024  __alloc.c_0
...
400001E9 00000134  errno
...
400003AB 00000728  qd
...
```

In this example the statics contained in the file `alloc.c` reside at base address `40000000`. The TOC entry at offset `728` contains the address of `qd`.

If Generate cross references is turned on in the Symantec Project Manager's Linker **Project Options** dialog, then each entry in a section's content will be followed by a table of references to other entry points in the container. The format of the entry is: address of the reference followed by name of the reference. For example:

```
0000DE40 00000718 20001818   00000000  main
                             0000DE4C InitManagers(void)
                             0000DE56 __main.cp_0
                             0000DE7E __main.cp_0
                             0000DE8C NewCWindow
                             0000DE98 SetPort
                             0000DEB6 __main.cp_0
                             0000DECE __main.cp_0
                             0000DF40 HSV2RGB
                             0000DF4C RGBForeColor
                             0000DF80 PaintRect
                             0000DFB8 WaitNextEvent
```

In this example the `main` routine calls `SetPort` from address `DE98` and accesses statics (`__main.cp_0`) from addresses `DE56`, `DE7E`, `DEB6`, and `DECE`.

Most symbols in the link map have obvious origins. Various translators create special symbols (usually prefixed by '__') that appear in the link map.

Names in the form `__file.c_n` and `__file.cp_n` contain static data such as strings, floating-points, and routine descriptor values that are contained in the specified source file and user-declared static variables.

The name `_ptrgl` refers to the code that dispatches a routine via its routine descriptor. The names `__cplusstart`, `__cplusterm`, `__start`, and `__term` refer to initialization and termination routines.

Names in the form `__sinit__file.cp__ct_` and `__sterm__file.cp__ct_` refer to static constructors and destructors associated with the file. The names `_ctors` and `_dtors`

are generated by the linker to handle constructors and destructors. A name in the form `__vtbl__class` is the virtual function dispatch table for the specified class.

Symantec C++ Optimizer

An optimizer makes your code more efficient. The Symantec C++ optimizer can speed up your program considerably while making your program smaller.

Symantec C++ for Power Macintosh uses a global optimizer. When you compile your program, the compiler produces an internal representation. The optimizer edits the internal representation to make it quicker and smaller. The optimizer then passes this revised internal representation to the code generator, which actually produces machine instructions.

Why use an optimizer

Although you might be able to edit your own program, and you might even make it as efficient as an optimizer, by letting the optimizer edit it instead has these advantages:

- Time savings. Tuning up old code takes time away from writing new code.
- Legible code. To make code more efficient, you might need to rewrite it in a way that is hard to read.
- Portable code. Code that's tuned for one type of computer (such as a Macintosh) may be inefficient on another (such as an IBM PC).

When *not* to use an optimizer

Use an optimizer when you're making the final build of a program you plan to use several times. Here are some programs you *don't* want to optimize:

For more information, see the Symantec C++ User's Guide and Reference.

- A program used only a couple times. Using an optimizer may double your compilation time. It doesn't make sense to spend more time optimizing it than you'll spend using it.
- A program you're debugging. An optimizer generates machine code that differs significantly from your source code in execution order.

Calling Toolbox Routines 3

The Macintosh Toolbox is the set of over one thousand routines that give Macintosh applications a consistent interface. This chapter outlines what you have to do to call the routines described in *Inside Macintosh*.

Contents

Calling Toolbox Routines

With Symantec C++ for Power Macintosh, you can use all of the Macintosh Toolbox interfaces. To use the Toolbox interfaces, call them exactly as they appear in *Inside Macintosh*. The only thing you need to know is how to convert the Pascal declarations into C declarations.

Most Macintosh routines are implemented as calls that take Pascal-typed arguments.

To use a Macintosh Toolbox routine, you must be using the correct library and headers. If you're using `PPC MacHeaders` or `PPC MacHeaders++`, you don't have to include header files for the most common Toolbox routines.

Table 3-1 lists the Macintosh Toolbox libraries.

For documentation on HyperCard XCMDs and the Communications Toolbox, contact APDA.

This library...	Contains...
AppleScriptLib	AppleScript shared library
AppleScriptLib.xcoff	Stub for AppleScript shared library
DragLib.xcoff	Drag Manager interface
InterfaceLib.xcoff	Toolbox calls
MathLib.xcoff	Low-level floating-point math (required for PPCANSI.o library)
ObjectSupportLib	Apple event interface
ObjectSupportLib.xcoff	Apple event interface
PPCToolLibs.o	MPW tool library (includes a PowerPC disassembler)
QuickTimeLib.xcoff	Quick Time movie calls

Table 3-1 Macintosh Toolbox libraries

Note

The Symantec Linker allows you to add either the `.xcoff` stubs file or the equivalent shared library into your project.

Passing arguments to Toolbox routines

Since argument declarations in *Inside Macintosh* are given in Pascal, you need to know how to convert them to C. Table 3-2 gives you the general rule for converting argument declarations from Pascal to C:

If the object is...	Pass...
a `var` parameter	a pointer to the object
4 bytes or smaller	the object
larger than 4 bytes	a pointer to the object

Table 3-2 Rules for converting Pascal declarations

Table 3-3 lists some examples of Pascal declarations and their C and C++ counterparts:

Pascal type	C/C++ type
`integer`	`short`
`longint`	`long`
`char`	`short`
`Boolean`	`Boolean` or `char`
`Byte`	`Byte` in `struct` declarations, `short` when passed as an argument
`var Byte`	`short *`
`ProcPtr`	See "Working with Pascal strings" later in this chapter.
`Handle`	`Handle`
`var Handle`	`Handle *`
`Ptr`	`Ptr`
`var Ptr`	`Ptr *`
`OSType, ResType`	`OSType, ResType, long`
`packed array [1..4] of char`	`long`
`Str255`	`Str255` or `unsigned char *`
`var Str255`	`Str255` or `unsigned char *`
`StringPtr`	`StringPtr` or `unsigned char *`
`var StringPtr`	`StringPtr *` or `unsigned char **`
`Rect`	`Rect *`
`var Rect`	`Rect *`

Table 3-3 Examples of converting Pascal declarations

Pascal type	C/C++ type
`Point`	`Point`
`var Point`	`Point *`
`Extended`	`double`

Table 3-3 Examples of converting Pascal declarations *(Continued)*

Working with Pascal strings

Toolbox routines that take strings as arguments expect them to be Pascal strings. Unlike null-terminated C strings, Pascal strings begin with a length byte. To write a Pascal string constant, start the string with `"\p"`. This is how you would call the QuickDraw routine `DrawString()`:

```
DrawString("\pThis is a Pascal string");
```

Because Pascal strings start with a length byte, they cannot be longer than 255 bytes. They are *not* generally null terminated. By default, Pascal string literals are of type `unsigned char[]` to agree with the definition of `Str255`.

Note

The Symantec compilers always null terminate string constants.

You can use the routines in Table 3-4 to convert strings from one form to the other:

To convert a ...	Use the function...
C/C++ string to a Pascal string	`CtoPstr()`
Pascal string to a C/C++ string	`PtoCstr()`

Table 3-4 Functions that convert strings

These routines convert the strings in place and return the converted string. Their function prototypes are:

```
unsigned char *CtoPstr(char *s);
char *PtoCstr(unsigned char *s);
```

Note

If you're not using `PPC MacHeaders` or `PPC MacHeaders++`, include `pascal.h` to use these functions.

Using PPC MacHeaders/PPC MacHeaders++

PPC MacHeaders++ is a precompiled header containing the most common declarations for writing Macintosh C++ programs. Symantec C++ also includes PPC MacHeaders, which is the equivalent precompiled header for use with the C compiler.

Both precompiled headers can be found in the Mac #includes folder. If you include <PPC MacHeaders++> in the project prefix for Symantec C++, Symantec C++ includes PPC MacHeaders++ in the files in your project by default. Likewise, including <PPC MacHeaders> in the project prefix for Symantec C, places the definitions from PPC MacHeaders in your project's files. As a result, you never have to explicitly include common header files like QuickDraw.h. (It doesn't hurt if you do—the #pragma once directive prevents header files from being included more than once.)

Note

By default, the project prefix for Symantec C++ contains the line #include <PPC MacHeaders++> and the project prefix for Symantec C contains the line #include <PPC MacHeaders>.

To edit the project prefix, see "Creating your own precompiled header" in Chapter 2.

The PPC MacHeaders++ file and the PPC MacHeaders file contain these files:

AEObjects.h	AEPackObject.h
AERegistry.h	AppleEvents.h
AppleTalk.h	BDC.h
Components.h	ConditionalMacros.h
Controls.h	Desk.h
Devices.h	Dialogs.h
DiskInit.h	EPPC.h
Errors.h	Events.h
Files.h	FixMath.h
Fonts.h	GestaltEqu.h
IntlResources.h	Lists.h
LowMem.h	Memory.h
Menus.h	MixedMode.h
Notification.h	OSEvents.h
OSUtils.h	Packages.h
pascal.h	PPCToolBox.h
Processes.h	QuickDraw.h
QuickDrawtext.h	Resources.h
Scrap.h	Script.h
SegLoad.h	StandardFile.h
Strings.h	TextEdit.h
TextServices.h	TextUtils.h
THINK.h	Timer.h
ToolUtils.h	Traps.h
Types.h	Windows.h

The following files aren't used as often, so they're not included in either PPC MacHeaders++ or PPC MacHeaders. You can include them yourself, or make a custom version of

PPC MacHeaders++ or PPC MacHeaders, as described in "Precompiled Headers" on page 15.

```
ActionAtomIntf.h          ADSP.h
AIFF.h                    Aliases.h
AppleScript.h             Balloons.h
CommResources.h           Connections.h
ConnectionTools.h         CRMSerialDevices.h
CTBUtilities.h            DatabaseAccess.h
DeskBus.h                 Dictionary.h
Disks.h                   Editions.h
ENET.h                    FileTransfers.h
FileTransferTools.h       Finder.h
Folders.h                 Graf3D.h
HyperXCmd.h               Icons.h
ImageCodec.h              ImageCompression.h
Language.h                MediaHandlers.h
MIDI.h                    Movies.h
MoviesFormats.h           OSA.h
OSAComp.h                 OSAGeneric.h
Palette.h                 Palettes.h
Picker.h                  PictUtil.h
Power.h                   Printing.h
PrintTraps.h              QDOffScreen.h
QuickTimeComponents.h     Retrace.h
ROMDefs.h                 SANE.h
SCSI.h                    Serial.h
ShutDown.h                Slots.h
Sound.h                   SoundInput.h
Start.h                   SysEqu.h
Terminals.h               TerminalTools.h
Values.h                  Video.h
```

The Macintosh Header Files

Symantec C gets its information about the Toolbox routines from the header files in the Mac #includes folder. The header files contain C prototypes for all Toolbox routines.

The Mac #includes folder

The `Mac #includes` folder contains the folders and files described in Table 3-5:

Item	Description
`Universal Headers`	This folder contains all of the Macintosh header files from Apple.
`Mac #includes.c`	This file is the "source file" for `PPC MacHeaders`, the default precompiled header for Symantec C. Instructions in this file explain how to modify it to create a customized precompiled header.
`Mac #includes.cpp`	This file is the "source file" for `PPC MacHeaders++`, the default precompiled header for Symantec C++. It is used to select the compiler (C++) only and #includes `Mac #includes.c` for the definitions. See the "Customizing the PPC MacHeaders/PPC MacHeaders++ files" on page 15 for information on modifying this file.
`PPC MacHeaders`	This is the default precompiled header for Symantec C. It contains the symbols from the most commonly used Toolbox managers.
`PPC MacHeaders++`	This is the default precompiled header for Symantec C++. It contains the symbols from the most commonly used Toolbox manager.
`THINK #includes`	This folder contains header files unique to Symantec C/C++.
`Rez #includes`	This folder contains `include` files required by the resource compiler for building Macintosh resources.

Table 3-5 Files in the Mac #includes folder

Compiler Reference 4

This chapter describes in detail aspects of the Symantec C and C++ implementations that are not part of the C and C++ language definitions.

Contents

How Symantec Compilers Implement C and C++

Symantec C and C++ support the enhanced language features of version 3.0 of the C++ language including templates, nested classes, and nested types. Exception handling is not yet implemented.

Symantec C is very similar to earlier versions of THINK C. This section describes how Symantec C implements certain parts of the C language, including how it represents integers and floating-point numbers.

Identifier length and capitalization

Symantec C and C++ allow up to 1024 significant characters in an identifier. If you exceed this maximum, the compiler flags the identifier as a syntax error. Underscores, letters, and digits are allowed, and case is significant.

How Symantec Compilers Look for Header Files

These are the rules Symantec C and C++ use to find header files:

#include statement	**Symantec Compilers...**
`<filename.h>`	Look first in the referencing folder, then in the system tree.
`"filename.h"`	Look first in the referencing folder, then in the project tree, and finally in the system tree.

System tree refers to the folder containing the Symantec Project Manager application and all folders contained within it. Project tree refers to the folder containing the project file and all folders contained within it.

The referencing folder is the folder that contains the file that has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h`, and that file in turn has the line `#include "MyUtilTypes.h"`, Symantec C and C++ look for `MyUtilTypes.h` first in the folder that contains `MyUtils.h`.

Once-only headers

You can create a header file that you want included in several places but that should define its symbols only once in a project. Use the `#pragma once` directive to do this.

For example, if you have the directive:

```
#pragma SC once
```

in your header file, Symantec C and C++ include that file only once. If another file tries to include that header file, the compilers know that the symbols in that file have already been defined, so the file is not processed again.

Note

Placing the SC in the directive forces the compilers to produce an error if the directive is not recognized and is not required.

Keep the following restrictions in mind when you use `#pragma once`:

- It doesn't distinguish between files included with `<...>` and `"..."`. For example, suppose you have two header files named `xyz.h`: one in the system tree and one in the project tree. If you include one with `#include <xyz.h>` and another with `#include "xyz.h"`, the compiler will not include the second file.

- It ignores characters after the first 32. If you include two files with names that start with the same 32 characters, neither compiler will include the second file.

Shielded folders

To shield a folder from either search tree, enclose its name in parentheses. For example, you might have a folder in the project folder named `(Backups)`. Symantec C and C++ ignore all files and subfolders in shielded folders. You can use shielded folders to store old versions of source or header files or to keep Symantec C and C++ from wasting time looking in folders that contain other kinds of documents such as development notes.

Project-specific folders

There is one exception to the shielding rule. If the folder your project is in contains a folder that has exactly the same name as your project surrounded by parentheses, Symantec C and C++ will search that folder.

You can use this feature if you're working on two projects that share files. For example, if you're working on two projects, INITProject and cdevProject, that share some source files and are in the same folder, create two folders, `(INITProject)` and `(cdevProject)`. Both folders should contain a version of the header file `config.h` tailored to control conditional compilation of the common source files.

Using aliases

Symantec C and C++ let you work with the alias of a project file. The project tree begins where the original project is located. Also, you can place aliases to folders in your project and system trees, and those folders will be searched as though they exist within the tree containing the alias. Aliases to files are ignored.

Symantec C++ for Power Macintosh does not support aliases when:

- Putting aliases in a project
- Including aliases in an `#include` statement
- Using an alias as a project's resource (`.rsrc`) file

Using the trees

Symantec C++ for Power Macintosh lets you organize your files the way you like without having to specify full path names. There are a few points you should remember about using Symantec C++ and project trees.

Don't put project folders in the system tree

This is the most common mistake. It seems natural to put all your Symantec C or C++ files in one folder, then put your project folders in the folder as well. However, if you set up your disk this way, Symantec C++ searches all your other project trees every time it searches the system tree. Therefore, setting up your project folders this way not only increases search time, it increases the likelihood of duplicate names within trees.

Avoid duplicate file names in trees

You shouldn't have duplicate file names in different folders within the project or system tree. If you do, Symantec C++ won't know which file to use. Duplicate file names won't lead to any explicit errors, but you may end up using the wrong file.

It's okay to have the same file name in both the project and system trees. Symantec C++ resolves the conflict by search order.

Using Registers

Symantec C and Symantec C++ assign local variables to registers whenever possible. The PowerPC registers include 32 integer, 32 floating-point, and a few special purpose registers. Table 4-1 shows the usage conventions for these registers and their names.

Note

Registers designated as *volatile* in the table indicate the value is not preserved across function calls.

Register	Defined to be
GPR0	Scratch register *(volatile)*
GPR1	Stack pointer (SP) *(non-volatile)*
GPR2	TOC register *(non-volatile)*
GPR3 - GPR10	First 8 parameter words; also scratch registers; GPR3 is used for non-float results. *(volatile)*
GPR11, GPR12	Scratch registers; GPR11 is used for a function indirect call. *(volatile)*
GPR13 - GPR31	*(non-volatile)* registers
FPR0	Scratch register *(volatile)*
FPR1 - FPR13	First 13 floating-point parameters; also scratch registers; FP1 and FP2 are used for float results. *(volatile)*
FPR14 - FPR31	*(non-volatile)* floating-point registers
CR	Condition code register contains 8 fields, CR0 - CR7. CR0 is set by default by some integer operations. CR1 is set by default by some floating-point operations. The others could be used at the code generator's discretion. *(volatile)*
LR	Link register
CTR	Counter register, (loops, computed branches) *(volatile)*

Table 4-1 Register assignments

The compilers do not place register variables into any of the special purpose registers. The policy used to allocate variables to registers depends on whether or not the function is a leaf routine.

Leaf routines are routines that do not contain any function calls. In a leaf routine, any register can be used to contain register variables. In a non-leaf routine, non-volatile registers are preferred for register variables. However, variables that are needed only before or after a function call can also be placed into volatile registers.

Alignment of Structure or Array Members

Each data object has a type and a preferred alignment. An object's type is defined by the language, Symantec C or C++. Alignment mode, used to determine preferred alignment, defines how the members of an aggregate type reside in memory. Mode influences the size of an aggregate type and the offset of its members in memory. The current alignment modes are:

- powerpc
- mac68k

The default alignment mode for the PowerPC is Align to 4 byte boundary, and is controlled by the Compiler Settings page of the **Project Options** dialog. To change the alignment mode, use the pragma:

For a detailed description of the #pragma align directive, see "#pragma Directives," later in this chapter.

```
#pragma options align=alignment-specifier
```

The preferred alignment of a data object is determined by the type and alignment mode in effect at the point in the source code where the aggregate type is defined. The alignment is determined by the byte boundary where the type is accessed most efficiently. On 68K machines the alignment is on 2-byte boundaries; on the PowerPC the alignment is on 4-byte boundaries and sometimes 8-byte boundaries.

Note

8-byte alignment boundaries are required when using double precision floating-point.

Table 4-2 shows the preferred alignment of aggregate data types for both powerpc and mac68k modes.

Data Type	powerpc Mode	mac68k Mode
char	1	1
short	2	2
int	4	4
pointer	4	4
float	4	4
double	4	4
long double	4	4
enumeration	Preferred alignment of same sized integer type	Preferred alignment of same sized integer type
array	Preferred alignment of member type	Preferred alignment of member type
union	8, if any member is a double or long double, *or* largest of preferred alignments of the member types	2
struct	8, if first member is a double or long double or an aggregate with preferred alignment of 8, *or* 4, if largest preferred alignment of remaining member types is 4, *or* largest of preferred alignments of the member types	2

Table 4-2 Preferred alignment of aggregate data types

Integer Representation

Integers are represented as two's complement binary numbers. The size of an int is 4 bytes. In C and C++, an int is signed by default.

Short integers

A short int is 2 bytes. The int in the declaration short int is optional and is usually omitted. In C And C++ a short is signed by default.

Long integers

A `long int` is 4 bytes. The `int` in the declaration `long int` is optional and is usually omitted. In C And C++ a `long` is signed by default.

Integer limits

This section describes limits for the Symantec C and C++ implementation of integers: the largest and smallest values for each integer type. These parameters are defined as macros in `limits.h`. Other implementations of C and C++ could have different values.

Changing the values of the macros won't change the way Symantec C or C++ represents integers. The values are listed in Table 4-3 to indicate the limits of the Symantec C and C++ integers.

Warning

Don't use the value of a macro in your source code. The value might change in future versions of Symantec C and C++. Use the macro's name instead.

Note

The natural integer type for the PowerPC is 4 bytes (`int` or `long`). Use this whenever possible since it will result in better code generation. Use shorter types only when space is very important or 68K and/or Toolbox compatibility is required.

Macro	Value	Description
`CHAR_BIT`	8	`Number of bits for the smallest object that is not a bit-field`
`MB_LEN_MAX`	1	`The maximum number of characters that can be in a multi-byte character`
`SCHAR_MIN`	-128	`Minimum value for a signed char`

Table 4-3 Integer parameters and limits

Macro	Value	Description
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	-128	Minimum value for a char
CHAR_MAX	127	Maximum value for a char
SHRT_MIN	-32,768	Minimum value for a short int
SHRT_MAX	32,767	Maximum value for a short int
USHRT_MAX	65,535	Maximum value for an unsigned short int
INT_MIN	-2,147,483,648	Minimum value for an int
LONG_MIN	-2,147,483,648	Minimum value for a long int
INT_MAX	2,147,483,647	Maximum value for an int
LONG_MAX	2,147,483,647	Maximum value for a long int
UINT_MAX	4,294,967,295	Maximum value for an unsigned int
ULONG_MAX	4,294,967,295	Maximum value for an unsigned long int

Table 4-3 Integer parameters and limits *(Continued)*

Floating-Point Representation

Use floating-point variables to store numbers that may have a fractional part. A floating-point number has two parts, a mantissa and an exponent. The size of the mantissa determines the number of digits of accuracy of the values you can store; the size of the exponent determines their range. Both of these are system-dependent. The size limits for the floating-point types are declared in the header file <float.h> and are documented in the online *Standard Libraries Reference.*

Symantec C++ for Power Macintosh gives you two types of floating-point numbers:

- `double` (8-byte IEEE double precision) is the fastest floating-point type. Use this type most of the time. Currently, `double` is the same size as a `long double`, however, that may change in future releases.
- `float` (4-byte IEEE single precision) is smaller than double. Use this type when you need to save space and can sacrifice some accuracy. It may also sacrifice code generation quality in some cases.

Floating-point parameters and limits

This section describes some limits and parameters for Symantec C and C++ floating-point numbers, such as the largest possible value and the smallest possible fraction. These parameters are defined as macros in the header file `float.h`.

Changing the values of the macros doesn't change the way Symantec C and C++ represent floating-point numbers. The values are listed in the following tables to let you know the limits and attributes of the numbers in Symantec C and C++.

Warning

Don't use the value of a macro in your source code. The value might change in future versions of Symantec C and C++. Use the macro instead.

For more information on how the Macintosh represents floating-point numbers on 68K machines, see the Apple Numerics Manual, Second Edition *(Addison-Wesley). For information on how the Macintosh represents floating-point numbers on the PowerPC, see the* Inside Macintosh PowerPC Numerics Manual, *(Addison-Wesley).*

The macros in Table 4-4 define attributes for the Symantec implementation of floating-point numbers.

Macro	Description
`FLT_RADIX`	The radix for exponent representation. Symantec C and C++ use 2.
`FLT_ROUNDS`	Specifies how floating-point functions round the results of addition. Symantec C and C++ round to the nearest floating-point value.

Table 4-4 Attributes of Symantec C and C++ floating-point numbers

The macros in Table 4-6 define limits and parameters for the Symantec C and C++ implementation of floating-point numbers. The first few letters of the macro indicate the type of numbers for which the macro is used, as in Table 4-5.

Macro	Value	Description
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 53 53	The number of base FLT_RADIX digits in the floating-point significand
FLT_DIG DBL_DIG LDBL_DIG	6 15 15	Decimal digits of precision
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	-125 -1021 -1021	The largest integer that can be a negative exponent when the radix is FLT_RADIX
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	-125 -307 -307	The largest integer that can be a negative exponent when the radix is FLT_RADIX
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 1024 1024	The largest integer that can be a positive exponent when the radix is FLT_RADIX
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 308 308	The largest integer that can be a positive exponent when the radix is FLT_RADIX
FLT_MAX DBL_MAX LDBL_MAX	3.40282E+38 1.79769E+308 1.79769E+308	The largest representable floating-point number
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209E-7 2.22045E-16 2.22045E-16	The smallest fraction that can be represented in floating-point notation
FLT_MIN DBL_MIN LDBL_MIN	1.17549E-38 2.22507E-308 2.22507E-308	The smallest normalized positive floating-point number

Table 4-5 Floating-point parameters and limits

Macros starting with...	Describe limits for...
FLT	float
DBL	double
LDBL	long double

Table 4-6 Floating-point macro prefixes

Unordered Comparisons

The Symantec C and C++ compilers allow the use of the unordered, floating comparisons operators defined as:

!<>=	Unordered
<>	Less than or greater than
<>=	Not unordered (less than, equal to, or greater than)
!<=	Not less than or equal to (unordered or greater than)
!<	Not less than (unordered, greater than, or equal to)
!>=	Not greater than or equal to (unordered or less than)
!>	Not greater than (unordered, less than, or equal to)
!<>	Unordered or equal

For more information on NaNs and unordered comparisons, see the Inside Macintosh PowerPC Numerics Manual, *(Addison-Wesley).*

These unordered comparison operators allow for either or both operands to be a NaN (Not-a-Number). For example, taking the square root of a negative number yields a NaN. Other numbers cannot be compared successfully with this NaN for any of the ordered comparison operators (<, <=, >, >=).

The unordered comparison operators allow control over the case when a number is compared to a NaN. In general, a NaN yields a FALSE result for any of the normal comparison operators (although this depends on the specific NaN and the other number). To test whether two numbers can be compared using <, <=, >, or >=, use the <>= operator as in:

```
X = f(Y);
if (Y <>= X) {     //neither is a NaN
   if (Y < X)
      .
      .
}
```

In general, to yield TRUE if one operand is unordered, use `(! (X !< Y)` instead of `X < Y`, and so on.

Dimensionless Arrays

You can use dimensionless arrays as the last member of `struct` definitions. The member does not contribute to the size of the `struct`. For example:

```
struct {
   int count;
   char data[];
} CountData;    /* sizeof(CountData) is 4*/
```

The _new_handler

(C++ only) The `_new_handler` variable lets you call one of your functions if a call to new fails due to lack of memory. The program can then use the function to free up more memory. If you use `_new_handler`, you don't always need to check the return value of new for failure.

`_new_handler` is a pointer to a function. It is declared in the CPlusLib library, and is set to `NULL` by default. Its declaration is:

```
void (*_new_handler)(void);
```

When new fails, it tests if `_new_handler` points to a function or if `_new_handler` is `NULL`. If `_new_handler` contains a value, the function it points to is called. If `_new_handler` is `NULL`, new returns a `NULL` pointer. You must set `_new_handler` explicitly. You can set `_new_handler` directly as:

```
void newfailed_handler(void);
      // prototype of handler
_new_handler = newfailed_handler;
      //set _new_handler
```

or through the `set_new_handler` library function:

```
set_new_handler(newfailed_handler);
```

Note

You must `#include <new.h>` to make these declarations.

The Inherited Keyword

(C++ only) Use the `inherited` keyword to access a base-class version of a member function without explicitly naming the base class. For example:

```
inherited::functionname
```

refers to the instance of *functionname* that the compiler would have found if *functionname* had not been declared in the current class.

The `inherited` keyword is not a portable language extension.

Note

If there is more than one direct base class, the first one declared is used.

Internal Limits

Table 4-7 specifies how big you can make certain aspects of your code.

Description	Limit
Characters in a line	No limit
Characters in an identifier	1024†
Characters in an external identifier	1024†
Characters in a string	No limit
Number of cases in a switch	No limit
Characters in an argument to a macro	No limit
Number of arguments to a macro	No limit
Number of arguments to a function	No limit
Length of macro replacement text	No limit
Number of subscripts in an array	No limit
Complexity of a declaration	No limit
Number of `#includes` that can be nested	No limit
Number of `#include` paths	No limit
Number of `#ifs` that can be nested	No limit
Number of command line arguments	No limit

Table 4-7 Internal limits of Symantec C and C++ implementations

† In C++, the 1024 limit applies to the object's full signature including the type.

Note

"No limit" means that the compiler establishes no limit. The operating system or the amount of memory available to the compiler may impose a practical limit.

Symantec C++ for Power Macintosh Extensions

The Symantec C and C++ compilers contain language extensions that let you program the Macintosh more easily. The compilers also have extensions that let you compile code written for less strict language implementations. These extensions do not conform to ANSI restrictions on the C or C++ languages. This section describes these extensions and how you can remove them to make your code more portable.

Strict ANSI C conformance

If you check the Strict ANSI conformance option on the Language Settings page of the **PowerPC C** compiler, the compiler adds the following ANSI-compatible restrictions to the Symantec C language:

1. You cannot use two slashes (`//`) to introduce a comment.

2. You cannot use dimensionless arrays as the last member of `struct` definitions. For a description of this feature, see "Dimensionless Arrays" on page 50.

3. The following keywords are not recognized:

`asm`	`__pasobj`
`cdecl`	`_cdecl`
`__handle`	`__fortran`
`inherited`	`_inf`
`__nans`	`__machdl`
`overload`	`__nan`
`pascal`	`_pascal`

4. You cannot use the unordered, floating comparisons operators, `!<>=`, `<>`, `<>=`, `!<=`, `!<`, `!>=`, `!>`, and `!<>`. See "Unordered Comparisons" on page 49 for a description of these operators.

5. You cannot have empty `struct` definitions.

6. You cannot use arithmetic on pointers to functions.

7. Trigraphs are supported. Trigraphs are sequences of three letters that are treated as one. The sequence is `??` and an additional character. Trigraphs let computers without such characters as braces (`{,}`), tildes (`~`), and carets (`^`) use C++. However, many Macintosh applications use character literals that resemble trigraphs. For example, the file type `'????'` is interpreted as `'?^`. To write the file type `'????'`, use `'???\?'`.

8. Text on the end of a preprocessor line is not ignored and is an error. For example, you would need to change `#endif COMMENT` to `#endif /*COMMENT*/`.

9. Empty member lists in `enum` declarations and member lists with a trailing comma are syntax errors.

10. You cannot use binary numbers such as `0b10110`.

11. At least one hexadecimal number must follow a `\x` escape sequence.

12. The program must end with a newline. Each translation unit must end with a newline.

13. You cannot obtain the size of a function with `sizeof`.

14. You cannot use hexadecimal floating-point constants.

15. A non-integer expression is not converted to an integer expression where a constant expression is required.

16. You cannot put a `sizeof` or a `cast` in a preprocessor expression.

17. The comma (`,`) operator is not allowed in constant expressions.

18. You cannot place a `void` expression in a logical `&&` or `||`. For example,

```
void  f();
   a || f();   //error: voids have no value
```

19. A function declared with ellipsis (`...`) does not match a function declared without ellipsis.

20. Old style function definitions must match the promoted size of the arguments. For example,

```
void  f(char);

void  f(c)
char  c;     //error: Prototype for f
             should be f(int);
{

}
```

21. Function definitions must match the number of prototyped arguments.

22. `case` label constants must be of type `int` or `unsigned int`.

23. Declarators must declare at least one variable.

24. Function prototypes with ellipsis must have at least one other argument.

25. Macros must match exactly when being redefined.

26. Unrecognized # directives are in error.

Relaxed ANSI C conformance

The Relaxed ANSI conformance option on the Language Settings page of the **PowerPC C** compiler maintains the restrictions in the above list except numbers 1, 2, and 8.

The Strict Prototype Enforcement option lets you choose how strictly Symantec C enforces the use of prototypes. If this option is on, you can choose between two enforcement levels: Infer prototypes and Require prototypes. If this option is off, Symantec C does nothing when you use or define a function without a prototype. To be ANSI-conformant, turn on Strict Prototype Enforcement and Infer prototypes.

Strict ANSI C++ conformance

If you check the Strict ANSI conformance option on the Language Settings page of the **PowerPC C++** compiler, the compiler adds the following ANSI-compatible restrictions to the Symantec C++ language:

1. These keywords are not recognized:

`asm`	`__pasobj`
`cdecl`	`_cdecl`
`__handle`	`__fortran`
`inherited`	`_inf`
`__nans`	`__machdl`
`overload`	`__nan`
`pascal`	`_pascal`

2. You cannot use arithmetic on pointers to functions.

3. Trigraphs are supported. Trigraphs are sequences of three letters that are treated as one. The sequence is `??` and an additional character. Trigraphs let computers without such characters as braces ({,}), tildes (~), and carets (^) use C++. However, many Macintosh applications use character literals that resemble trigraphs. For example, the file type `'????'` is interpreted as `'?^`. To write the file type `'????'`, use `'???\?'`.

4. Text on the end of a preprocessor line is not ignored and is an error. For example, you would need to change `#endif COMMENT` to `#endif /*COMMENT*/`.

5. Empty member lists in `enum` declarations and member lists with a trailing comma are syntax errors.

6. You cannot use binary numbers such as `0b10110`.

7. At least one hexadecimal number must follow a `\x` escape sequence.

8. The program must end with a newline. Each translation unit must end with a newline.

9. You cannot obtain the size of a function with `sizeof`.

10. You cannot use the unordered, floating comparisons operators, `!<>=`, `<>`, `<>=`, `!<=`, `!<`, `!>=`, `!>`, and `!<>`. See "Unordered Comparisons" on page 49 for a description of these operators.

11. You cannot use hexadecimal floating-point constants.

12. You cannot cast an `lvalue` to a different type.

13. Anonymous unions must be static.

14. A non-integer expression is not converted to an integer expression where a constant expression is required.

15. Member functions cannot be static. For example, the following declaration is legal C++:

```
    class Foo {
    public:
      void static int f(void) {return 3;}
    int b(void);
};
```

However, the following definition is a compile-time error:

```
    static int Foo::b(void)
{
      return 1017;
}
```

16. A reference cannot be generated to a temporary.

17. The type `void *` is not compatible with other pointer types.

18. You cannot convert to and from a `void`.

19. You cannot type something `void` where a value is required.

20. You cannot put a `sizeof` or a `cast` in a preprocessor expression.

21. You cannot use the pre-increment or post-increment operator function as an overloaded function for post increment or post decrement.

22. You cannot use dimensionless arrays as the last member of `struct` definitions. For a description of this feature, see "Dimensionless Arrays" on page 50.

23. You cannot use `#ident`.

24. Function declarations with separate parameter lists never match functions with supplied prototypes.

25. The comma (`,`) operator is not allowed in constant expressions.

26. You cannot place a `void` expression in a logical `&&` or `||`. For example,

```
void  f();
   a || f();   //error: voids have no value
```

27. Function definitions must match the number of prototyped arguments.

28. `case` label constants must be of type `int` or `unsigned int`.

29. String literals used to initialize arrays of `char` are always null terminated.

30. Declarators must declare at least one variable.

31. Function prototypes with ellipsis must have at least one other argument.

32. Function definitions with separate parameter lists are disallowed.

33. Template class instantiations cannot introduce new non-local names. For example,

```
template <class T>   class X  {
public:  int   i;
   friend   int   operator == (const X &x,
                               const X &y)
                    {
                       return x.y == y.i;
                    }
};

X <int> i;      //error: Non-local name
                operator == (const x&,
                const x&) cannot be
                declared in a template
                instantiation
```

34. Macros must match exactly when being redefined.

35. Unrecognized # directives are in error.

Relaxed ANSI C++ conformance

The Relaxed ANSI conformance option on the Language Settings page maintains the restrictions in the above list except numbers 1, 3, 12, and 22. These items are of specific relevance to Macintosh programming. Use the Relaxed ANSI conformance option to ensure strict type checking while retaining the extensions necessary for Power Macintosh programming.

Predefined Macros

Symantec C++ for Power Macintosh predefines the macros listed below.

Note

In the context below, the name One means the preprocessor expansion `1`.

__SC__, SYMANTEC_C, SYMANTEC_CPLUS, __ZTC__, SC_PLUS_SYMANTEC
The hex version number of Symantec C++. The current version is 0x800. `SYMANTEC_CPLUS` is defined only for the integrated Symantec C++ translator (PowerPC C++), to distinguish it from the Symantec C++ for MPW translator. All Symantec C and C++ compilers define `__SC__`.

macintosh, MC601, mc601
One.

__cplusplus
One in Symantec C++; undefined in Symantec C.

__LINE__
The current line number in the source file.

__FILE__
Set to a string containing the name of the current source file.

__DATE__
Set to a string containing the current date.

__STDC__
Defined as One when using ANSI conformance (relaxed or strict) in the C compiler.

__TIME__
Set to a string containing the current time.

__POWERC, powerc, __powerc
One.

__FPCE__, __FPCE_IEEE__
One, to indicate support for NCEG and IEEE conformance.

Note

__STDC__ is defined (it indicates ANSI C conformance) when using the Symantec C compiler; `applec` is never defined (Symantec C++ for MPW defines it to be one).

#pragma Directives

Symantec C++ for Power Macintosh implements the following `#pragma` directives:

```
align           once
export          options
external        parameter
import          segment
internal        template (C++ only)
lib_export      template_access (C++ only)
message         trace off
noreturn        trace on
```

The Symantec C++ for Power Macintosh compilers ignore the following pragmas: `parameter`, `segment`, `trace on` and `trace off`; they are maintained for 68K compatibility.

The pragmas are case-sensitive. Pragma directives are in the form:

`#pragma [SC]` *pragma-directive* [*pragma_args*]

If you specify `SC`, the pragma directive must be one of the pragmas recognized by the Symantec C or C++ compilers. If you do not specify `SC` and the pragma directive is not one of those listed above, then the Symantec C or C++ compiler assumes that the pragma is for another compiler and produces a warning.

#pragma [SC] align
This pragma lets you set byte alignments within structures. It takes the form:

`#pragma [SC] align` *[1|2|4]*

The optional number indicates which byte boundary to align on. The default is 4, which maximizes performance on systems with a 32-bit bus. If you use this pragma with no argument, the compiler uses the default setting. This option is for backwards compatibility. Use 2 for mac68k and use 4 for PowerPC.

Note

The `#pragma options align` syntax is preferred.

#pragma [SC] export

This directive informs the compiler which functions or data items should be exported from the fragment being built. The pragma has the following syntax:

```
#pragma export on
#pragma export off
#pragma export list name1 [ ,  name2 ]*
```

Any items declared or defined while export is *on* or specified by name through the export list are tagged as exported items and, if present in the linkage unit, will be exported by name from the code fragment (the resulting PEF container).

#pragma [SC] external

This is the default status for global variables and indicates that a global variable is available to be exported. The pragma has the following syntax:

```
#pragma external on
#pragma external off
#pragma external list name1 [ ,  name2 ]*
```

The code generator cannot perform any optimization that will prevent this symbol from being exported from the fragment. This can be used with the linker's Export all symbols option. For more information, see the *Symantec C++ User's Guide and Reference.*

#pragma [SC] import

This pragma informs the linker that a specified symbol should be called using a glue function even within its own fragment. Without this #pragma in effect, a symbol cannot be patched by subsequent loading of a patch fragment because internal calls will not be made

through glue. The compiler must generate all calls to this function as cross TOC calls. For more information, refer to the *Inside Macintosh PowerPC System Software Manual* (Addison-Wesley).

This directive has the following syntax:

```
#pragma import on
#pragma import off
#pragma import list name1 [ ,  name2 ]*
```

Any items declared or defined while import is *on* or specified by name through the import list are tagged as imported items and get cross fragment code generation for references to them.

#pragma [SC] internal

This pragma allows the compiler to generate better code by assuming that calls to the specified functions will always be local calls. It has the following form:

```
#pragma internal on
#pragma internal off
#pragma internal list name1 [ ,  name2 ]*
```

Any items declared or defined while internal is *on* or specified by name through the internal list are tagged as internal items.

This pragma can be used to tell the compiler that a function with external visibility is guaranteed not to be exported by name from its linkage unit (this would, in the absence of this directive, be quite permissible within the semantics of C or C++).

Incorrect usage of this pragma (i.e., turned on by the developer to improve code generation, when the function really is exported from the fragment) is detected at link time by way of a missing symbol.

#pragma [SC] lib_export

The effect of `#pragma lib_export` is to export symbols from a code fragment. Any symbols mentioned in the `#pragma` will be exported from the code fragment when it is built. Any calls to function symbols mentioned in the pragma's list will be made through glue code even within the fragment. The syntax is:

```
#pragma lib_export [ on | off | list <name> [ , ... ] ]
```

#pragma [SC] message

This pragma causes the compiler to print the specified text as a warning message while compiling. The syntax is:

```
#pragma [SC] message "text"
```

#pragma [SC] noreturn(function-name)

This pragma informs the compiler that the function does not return, which enables the compiler to generate improved code. The syntax is:

```
#pragma [SC] noreturn( identifier )
```

This pragma is useful for marking functions such as `exit()`, `_exit()`, `abort()`, `longjmp()`, and especially `assert()`, which never return to the caller.

#pragma [SC] once

When this directive appears in a header file, Symantec C++ includes the file only once even if `#include` directives include it multiple times.

```
#pragma [SC] once
```

Note

Any file included in a precompiled header will only be included once, whether or not this `#pragma` is specified.

#pragma [SC] options

This pragma directive lets you change options specified in the **PowerPC C** or **PowerPC C++** options and **Project Type** dialogs.

The `#pragma options` directive is described in "Accessing Option Settings in Your Code," beginning on page 67.

#pragma [SC] options align

This pragma lets you set byte alignments for 68K or the PowerPC. It takes one of the forms:

```
#pragma [SC] options align=power
#pragma [SC] options align=native
   (equivalent to #pragma [SC] align 4)

#pragma [SC] options align=mac68k
   (equivalent to #pragma [SC] align 2)

#pragma [SC] options align=reset
   (equivalent to #pragma [SC] align)
```

For PowerPC alignment, use "power" or "native." Use "mac68k" for alignment on 68K machines. "reset" reverts the alignment mode back to that of the second most recent alignment pragma, or to the default mode if none was specified. Modes may be arbitrarily nested. A data object's alignment mode is the mode in effect when the aggregate type is defined.

#pragma [SC] parameter

This directive is accepted for backwards compatibility, but is ignored on the PowerPC.

#pragma [SC] segment

This directive is accepted for backwards compatibility, but is ignored on the PowerPC.

#pragma [SC] template

This pragma produces one or more instantiations of a template in a source file. It uses the following syntax:

```
#pragma [SC] template class<arg1, arg2, ...>
#pragma [SC] template function(arg1, arg2, ...)
```

You can use these pragmas anywhere in your source file to expand the specified templates at the end of the file. It does not matter where the pragma occurs in relation to the template declaration. One typical use is after a `#include` directive that specifies the interface or source file for the template.

For example, assume file `vector.cp` contains the following:

```
template<class T> class vector {

T* v;
   int size;
public:
   vector(int);
   T& operator[] (int);
   // other
};

#pragma template vector<int>
 // will instantiate a vector class for
ints.
#pragma template vector<double>
 // will instantiate a vector class for
// doubles.
```

Note

This `#pragma` will expand only the specified template and not any templates that it depends on.

#pragma [SC] template_access

The template_access pragma option controls the access of template expansions that occur during compilation. The three types of access—`public`, `extern`, and `static`—allow flexibility in template instantiation. The syntax is:

```
#pragma [SC] template_access public

#pragma [SC] template_access extern

#pragma [SC] template_access static
```

Public access means that templates' names are globally accessible. In public access mode, templates are not expanded unless specifically mentioned in a `#pragma template` directive as described earlier in this section. If one template is expanded with public access in two different files, you get a link error in Symantec C++. Public access is most useful when used with external access.

External access means that templates specified are not expanded during compilation. Instead, the compiler generates an external reference to any name it would normally expand. This is useful when several source files use a particular template but you need

only one copy of it. You can designate one source file as your "template expansion" file, use public access for it, and then use extern access for all other source files.

Static access means that templates are expanded as usual, but their names are local to the current source file. This is useful for projects where you don't want to leave a special file aside for expanding templates. It is the default setting for Symantec C++.

Note

Static access cannot be used for template classes containing static data that are used in multiple translation units, or for template classes containing virtual functions.

It is an error to declare and use a template without defining it in a translation unit where `#pragma template_access static` is in effect. For example,

```
template <class T>   Min(T t1, T t2);
.
.
.
Min(1, 2);
.
.
.                    //error: static
                     Min (int, int) not
                     defined.
```

For an example on how to use these directives, see the Vector tutorial in the *Symantec C++ User's Guide and Reference.*

#pragma [SC] trace on
This directive is accepted for backwards compatibility, but is ignored on the PowerPC.

#pragma [SC] trace off
This directive is accepted for backwards compatibility, but is ignored on the PowerPC.

Accessing Option Settings in Your Code

You can use preprocessor directives to change and test most of the options specified in the **PowerPC C** or **PowerPC C++** options and **Project Type** dialogs. The `__option` directive lets you query option settings and the `options` pragma directive lets you change them. Both directives take option names as arguments, which are described later in this section.

Note

There are two underscores in `__option`.

To see if an option is enabled, use the `__option` directive in a `#if` directive. For example, in the following code fragment, the macro `OPTIMIZING` is set according to how you set the global optimizer option:

```
#if __option(global_optimizer)
   #define OPTIMIZING 1
#else
   #define OPTIMIZING 0
#endif
```

To change the setting of an option, use the `options` pragma directive. It takes any number of option names, separated by commas. To turn an option on, include its name in the list. To turn an option off, put an `!` in front of its name. For example, this code fragment turns on the global optimizer and turns off the ANSI conformance option:

```
#pragma options(global_optimizer,!ansi)
```

You can place an `options` pragma anywhere in your file. If it appears outside a function, it applies for the remainder of the file. If it appears inside a function, it applies to that entire function only. The previous settings are restored when the compiler finishes compiling that function. You can change only certain options inside a function.

Note

If an option is set more than once in a function, only the last setting is honored.

Options not applicable to Symantec C++

All of the option settings recognized by THINK C and prior versions of Symantec C++ are recognized. Options not implemented return a constant value of 1 and are not settable. The following options are not applicable to Symantec C++ for Power Macintosh:

```
a4_globals              macsbug_names
align_arrays            mc68020
assign_register         mc6881
class_names             mc6881_trans
defer_adjust            native_fp
double_8                objectc
far_code                pcrel_strings
far_data                profile
gopt_coloring           redundant_loads
gopt_induction          separate_strs
honor_register          signed_pstrs
indirect                thinkc
int_4                   trigraphs
jump_table              virtual
long_macsbug_names
```

Language extension options

The options, as specified in Table 4-8, control whether you can use some Symantec C++ for Power Macintosh extensions to the C and C++ languages. They correspond to options on the Language Settings page of the **PowerPC C** and **PowerPC C++** options dialog. You can test and set these options.

If this is set...	Then Symantec C++...
`stdc`	Returns 1 if Strict ANSI C or Relaxed ANSI C is set in PowerPC C; returns 0 in PowerPC C++.
`objectc`	Returns 1 if using PowerPC C++; returns 0 if using PowerPC C.
`ansi`	Corresponds to ANSI Settings button on Language Settings page of compiler options dialog.

Table 4-8 Language extension options

If this is set...	Then Symantec C++...
`ansi_relaxed`	Corresponds to Relaxed ANSI conformance button on Language Settings page of compiler options dialog.
`ansi_strict`	Corresponds to Strict ANSI conformance button on Language Settings page of compiler options dialog.

Table 4-8 Language extension options *(Continued)*

Enumerated type option

This option, as specified in Table 4-9, lets you choose what size enumerated types can be. It corresponds to the enums are always ints option on the Language Settings page of the **PowerPC C** and **PowerPC C++** options dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`pack_enums`	Enumerated types can be the size of any integral type.

Table 4-9 Enumerated type option

Include header once option

This option, as specified in Table 4-10, lets you include a header file once if it begins with a `#ifdef` and ends with a `#endif`. It corresponds to the Read each header file once option on the Language Settings page of the **PowerPC C** and **PowerPC C++** options dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`read_header_once`	`#if ... #endif` enclosed header files treated as if they contain `#pragma [SC] once`.

Table 4-10 Include header once option

Treat chars unsigned option

This option, as specified in Table 4-11, lets you treat objects declared as `char` as if they were declared as `unsigned char`. It corresponds to the treat chars as unsigned option on the Language

Settings page of the **PowerPC C** and **PowerPC C++** options dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`chars_unsigned`	`char` is unsigned; otherwise, it is signed by default.

Table 4-11 Treat chars unsigned option

Map carriage returns option

This option, as specified in Table 4-12, replaces all occurrences of `\n` (`0xa`) in character string literals with `\r` (`0xd`) and all occurrences of `\r` (`0xd`) become `\n` (`0xa`). This option must be on when linking with the Apple PowerPC libraries and off when linking with the Symantec standard libraries. The option corresponds to the Map carriage returns option on the Language Settings page of the **PowerPC C** and **PowerPC C++** options dialog. You can change it outside a function, but not inside one.

If this is set...	Then...
`mapcr`	`\n` (`0xa`) in character strings replaced with `\r` (`0xd`) and `\r` (`0xd`) replaced with `\n` (`0xa`).

Table 4-12 Map carriage returns option

Type-checking options

These options, as specified in Table 4-13, let you control how strictly PowerPC C type-checks your code. They correspond to options on the Language Settings page of the **PowerPC C** options dialog. You can change them outside a function, but not inside one.

If this is set...	Then PowerPC C...
`check_ptrs`	Checks pointer types. This option has the same effect as turning on the Check pointer types option. (PowerPC C++ returns 1.)
`require_protos`	Requires function prototypes. (PowerPC C++ returns 1.)
`infer_protos`	Infers a prototype when a function is first used. (PowerPC C++ returns 0.)

Table 4-13 Type-checking options

With `require_protos` and `infer_protos` in PowerPC C, you can get one of the three possible settings for the Strict Prototype Enforcement option. Table 4-14 describes the option.

If require_ protos is...	And infer_ protos is...	You get this setting...
Off	Off	Strict Prototype Enforcement option off
Off	On	Strict Prototype Enforcement option on, and Infer prototypes option selected
On	On or Off	Strict Prototype Enforcement option on, and Require prototypes option selected

Table 4-14 Strict Prototype Enforcement option

Debugging options

These options, as specified in Table 4-15, help you debug your code. They correspond to options on the Debugging page of the **PowerPC C** and **PowerPC C++** options dialog. You can change them anywhere.

If this is set...	Then Symantec C++...
`force_frame`	Generates a stack frame for each function. This option has the same effect as turning on the Always generate stack frames option.
`stop_at_first_err`	Stops at first error in source file.
`report_all_err`	Reports all error in source file, or stops at first unrecoverable error.
`dont_inline`	Uses a function call for any inline functions.

Table 4-15 Debugging options

Global optimizer options

These options, as specified in Table 4-16, control the global optimizer, described in "Code optimization," in Chapter 5. They

correspond to options on the Debugging page of the **PowerPC C** and **PowerPC C++** options dialog. You can change them anywhere.

If this is set...	Then...
`global_optimizer`	Use the global optimizer on this code. This option is like turning on the Use global optimizer option.
`gopt_time`	If the global optimizer and this option are on, optimize for time; if the global optimizer is on and this option is off, optimize for space.

Table 4-16 Global optimizer options

Specific information and examples of each Warning option can be found in "Warning Messages" beginning on page 99. To find a particular Warning option, look it up by its pragma option name. For example, you can find information about generate_warn by its pragma option name in the "Enable warning message" section.

Warning options

These options, as specified in Table 4-17, help you isolate code that may not behave as expected. They correspond to options on the Warning Messages page of the **PowerPC C** and **PowerPC C++** options dialog. You can change them anywhere.

If this is set...	Then...
`generate_warn`	Produce warning messages.
`warn_unintended_assign`	Warning when conditional expression of a `for`, `if`, or `while` contains an assignment.
`warn_nest_comments`	Warn when C style comments are nested.
`warn_unused_expressions`	Warn when value of expression has not been used.
`warn_empty_loops`	Warn when semicolon appears immediately after `if`, `while`, or `switch`.
`warn_large_auto`	Warn when total size of automatic variables in procedure greater than 32KB.
`warn_old_style_delete`	Warn when using older style array `delete` operator. (PowerPC C++ only)

Table 4-17 Warning Message options

If this is set...	Then...
warn_missing_overloads	Warn when using postfix versions of ++ or -- instead of missing prefix ops; or when using prefix version instead of missing postfix ops. (PowerPC C++ only)
warn_ref_init	Warn when reference initialized with a temporary value. (PowerPC C++ only)
warn_used_before_set	Warn when attempt made to obtain uninitialized variable's value.
warn_return_addr_auto	Warn when address of auto variable is a function's return value
warn_unrecognized_pragma	Warn when #pragma not recognized.
warn_old_style_definition	Warn when using pre-ANSI function definitions with relaxed ANSI conformance. (PowerPC C++ only)
warn_cast_incomplete_type	Warn when pointer or reference to incomplete struct type is cast to pointer to another struct type. (PowerPC C++ only)

Table 4-17 Warning Message options *(Continued)*

Compiler Options Reference 5

This chapter describes the Symantec C++ for Power Macintosh Compiler Option dialogs and their equivalent AppleScript commands and pragmas. Within each page, commands are described in the order in which they appear.

Contents

Symantec C++ for Power Macintosh Compiler Options

Symantec C++ for Power Macintosh provides three ways to access most compiler options: the **Options** menu, AppleScript, and pragmas. This section describes the **Options** menu and AppleScript. It also includes a complete reference of using the three methods to access each compiler option.

The Options Menu

Use **Options** from the **Project** menu to choose compiler options. Select the desired compiler from the scrolling list at the left of the **Project Options** dialog. There are six types of options, as shown in Figure 5-1. Each is described on one page of a single dialog box. To go to a certain page, select the appropriate name by clicking, scrolling with the arrow keys, or using the arrow buttons to the left of the pop-up menu. (To exit the Prefix page, press Command-up-arrow.)

Figure 5-1 The Options menu

The six types of compiler options are:

- **Language Settings** lets you choose extensions to the C and C++ languages.
- **Compiler Settings** lets you control how Symantec C++ for Power Macintosh generates code.
- **Code Optimization** lets you control how Symantec C++ for Power Macintosh optimizes your code.
- **Debugging** lets you specify how the Symantec debugger works.
- **Warning Messages** lets you control which warning messages (if any) Symantec C++ for Power Macintosh generates.

- **Prefix** lets you write code that Symantec C++ for Power Macintosh includes in all your files.

You can set options that affect only the current project, or set defaults that Symantec C++ uses whenever you create a new project. To set the options for use with all subsequently created projects, select **<New Projects>** from the **Options** menu at the top of the dialog. For each Option category, edit all the settings you want to change, and select Save.

When you click the Factory Settings button at the bottom of the page, Symantec C++ sets the options on all pages of the currently selected Options category to their original settings. For example, if PowerPC C++ is selected and you click on Factory Settings, the PowerPC C++ settings are restored to their original settings. No changes are applied to the other categories such as PowerPC C, Project Type, and so forth. All original settings are described in this chapter.

When you click Save, the program saves the changes for all pages of the dialog box.

AppleScript

For information on AppleScript, see the AppleScript Language Guide: English Dialect *(Addison-Wesley).*

All of the compiler options in the Symantec Project Manager are fully scriptable and recordable using AppleScript. Each setting in the **Compiler Options** dialog has a name and a set of possible values based on its type. For example, an AppleScript that turns off ANSI conformance in the Symantec C++ compiler would be:

```
tell application "Symantec Project Manager"
   activate
   set ansi of Options "PowerPC C++" of project
document "Poker Machine.π" to false
end tell
```

pragmas

Symantec C++ for Power Macintosh offers a comprehensive set of #pragma directives that give you an alternate way to access compiler options. See Chapter 4, "#pragma Directives," for complete information on using the `options` pragma.

C++ Language Settings

On the PowerPC C++ Language Settings page, you can choose whether or not the PowerPC C++ compiler uses extensions to the C++ language.

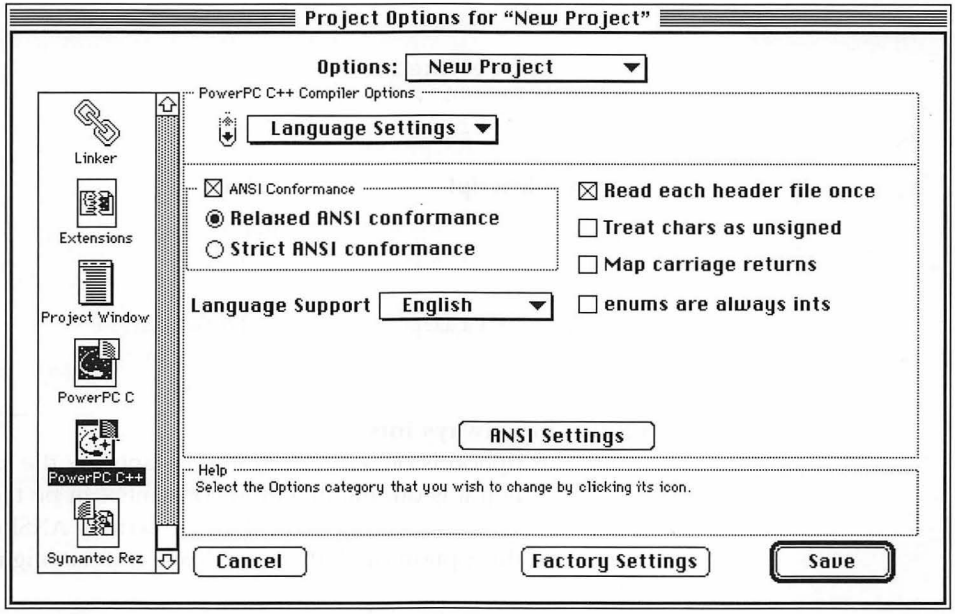

Figure 5-2 The PowerPC C++ Language Settings page

Use the options on this page to decide how closely the PowerPC C++ compiler conforms to the ANSI draft description of the C++ language. This is the only page of the C++ options that contains the ANSI Settings button at the bottom of the page. When you click that button, the options on this page are set to be strictly ANSI-conformant.

ANSI conformance

The ANSI conformance options are described in detail in the sections "Strict ANSI C++ conformance" and "Relaxed ANSI C++ conformance" in Chapter 4.

With this option originally on, you can choose between two levels of ANSI conformance in the PowerPC C++ compiler.

Relaxed ANSI conformance. This option is similar to strict ANSI-conformant, but it allows you to use language extensions that are convenient for Macintosh programming. The original setting is on.

Strict ANSI conformance. This option provides the strictest conformance to the ANSI draft specification. The original setting is off.

pragma option names

`ansi`

`ansi_relaxed`

`ansi_strict`

AppleScript	Values
`ansi`	`True` `False`
`ansi_strict`	`relaxed_ansi` `ansi_strict`

enums are always ints

When this option is on, enumeration constants are the same size as an `int`. When it is off, enumeration constants can be the same size as a `char`, `short int`, or `int`. If you're writing ANSI-conformant code, turn this option on. Otherwise, leave it in its original setting, off.

If this option is off, Symantec's PowerPC C++ makes enumeration constants as small as possible. And, if necessary, it makes a constant as large as an `int`. For example, these constants will only be as large as a `char`:

```
enum { red=1, yellow, green };
```

And these constants will be as large as an `int`:

```
enum {
   million=1000000, billion=1000000000
};
```

pragma option name

`pack_enums`

AppleScript	Values
`pack_enums`	`True` `False`

Read each header file once

If this option is on, PowerPC C++ treats header files that contain `#if ... #endif` around the entire contents of the file as if the file contained a `#pragma SC once` directive. Its original setting is on.

This option doesn't affect the meaning of the `#pragma once` directive.

pragma option name

`read_header_once`

AppleScript	Values
`read_header_once`	`True` `False`

Treat chars as unsigned

If this option is on, PowerPC C++ treats objects declared as `char` as if they were declared `unsigned char`. The types `char` and `unsigned char` are not equivalent types, even with this option on. The original setting is off.

pragma option name

`chars_unsigned`

AppleScript	Values
`chars_unsigned`	`True` `False`

Map carriage returns
If this option is on, PowerPC C++ replaces all occurrences of \n (0xa) with \r (0xd) in character constants and string literals. Use this option with the Apple-supplied libraries; turn it off with the Symantec-supplied libraries (see "Apple vs. Symantec standard libraries" on page 131). The original setting is off.

pragma option name

`mapcr`

AppleScript	Values
`map_cr`	`True` `False`

Language support
Use this option to accept foreign language double-byte characters in string and character literals, as well as comments. This option causes the compiler to produce localized error messages. Currently, any supported language may be used for input; however, Japanese is the only supported foreign language for error messages. The original setting is `english`.

pragma option name

(none)

AppleScript	Values

`native_language`	`english`
	`japanese`
	`chinese`
	`korean`

C Language Settings

On the PowerPC C Language Settings page, you can choose whether the PowerPC C compiler uses extensions to the C language.

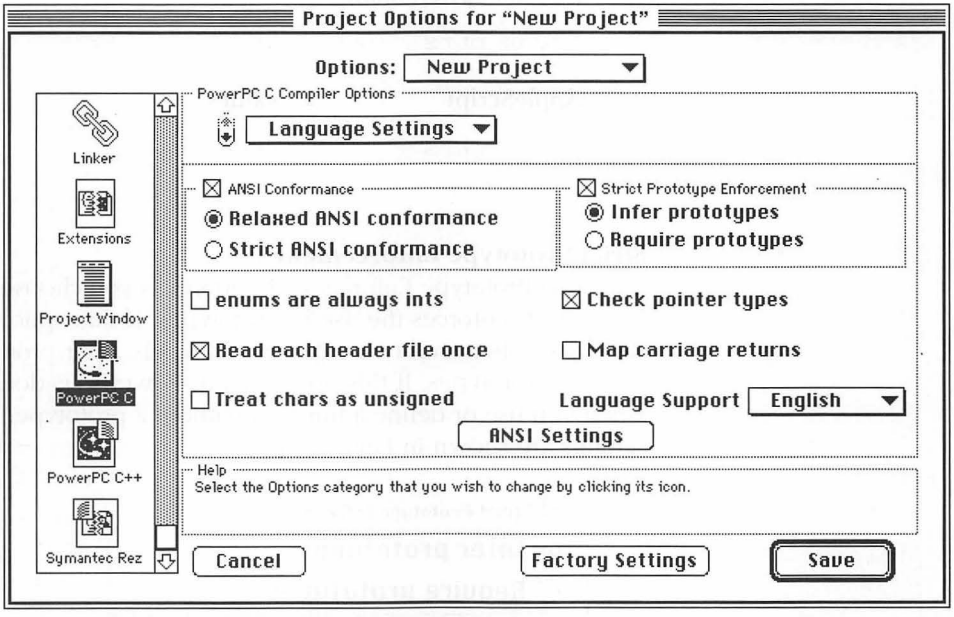

Figure 5-3 The PowerPC C Language Settings page

The options for PowerPC C are similar to those for PowerPC C++. The only differences are Check pointer types and Strict Prototype Enforcement. For descriptions of the common options, see the prior section.

Check pointer types

When the Check pointer types option is on, the PowerPC C compiler makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, Power PC C treats all pointers as equivalent types, and it won't

display the "Cannot implicitly convert" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size. The original setting is on.

pragma option name

`check_ptrs`

AppleScript	Values
`check_ptrs`	`True` `False`

Strict Prototype Enforcement

The Strict Prototype Enforcement option lets you choose how strictly PowerPC C enforces the use of prototypes. If this option is on, you can choose between two enforcement levels: Infer prototypes and Require prototypes. If this option is off, PowerPC C does nothing when you use or define a function without a prototype. The original settings are shown in Figure 5-4.

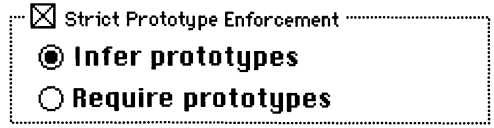

Figure 5-4 The Strict Prototype Enforcement option

Table 5-1 explains the two enforcement levels:

If you choose...	When you use a function without a prototype, PowerPC C...
Infer prototypes	Infers a prototype from the first appearance of a function. That appearance can be a function call or an old-style declaration. If a subsequent call, declaration, or prototype doesn't match the inferred prototype, it's an error.
Require prototypes	Raises an error. You can't use or define a function unless it has a prototype. New-style function definitions do not satisfy this requirement.

Table 5-1 Prototype enforcement levels

Note

There is one exception to the Require prototypes option requirement. A static function does not need a prototype. Just define it before using it.

When you use the Require prototypes option, PowerPC C does not perform argument promotion on an old-style function definition, if there is a prototype for it. Take this prototype and old-style definition for the same function:

```
int AFunction(char);

AFunction(b)
   char b;
{
   ...
}
```

If Require prototypes is not selected, PowerPC C promotes `b` to an `int`, and the prototype and definition don't match. If Require prototypes is selected, PowerPC C treats the old-style definition as a new-style definition, and the prototype and definition match.

These examples show how PowerPC C infers prototypes. If the first appearance of `PrintFloat()` is this call:

```
PrintFloat("Pi is ", 3.14159);
```

PowerPC C infers this prototype:

```
int PrintFloat( char *, double );
```

And if the first appearance of `PrintInt()` is this old-style declaration:

```
PrintInt( string, value )
   char *string;
   int value;
{
   printf( "%s%d\n", string, value );
}
```

PowerPC C infers this prototype:

```
int PrintInt( char *, int );
```

PowerPC C follows the rules of argument promotion when it infers prototypes. For example, it promotes parameters of type `char` and `short` to `int`, and parameters of type `float` to `double`. For example, from this function call:

```
PrintShort("This answer is: ", (short) 32);
```

PowerPC C infers this prototype:

```
int PrintChar( char *, int );
```

Note

PowerPC C promotes the `short int` to `int`, and infers that a function with no specified return type returns an `int`.

---

pragma option name

```
infer_protos
require_protos
```

AppleScript	Values
`infer_protos`	`infer_prototypes` `require_prototypes`

Compiler settings

The Compiler Settings page lets you control how the Symantec PowerPC C and PowerPC C++ compilers compile your code. The options are the same for both C and C++.

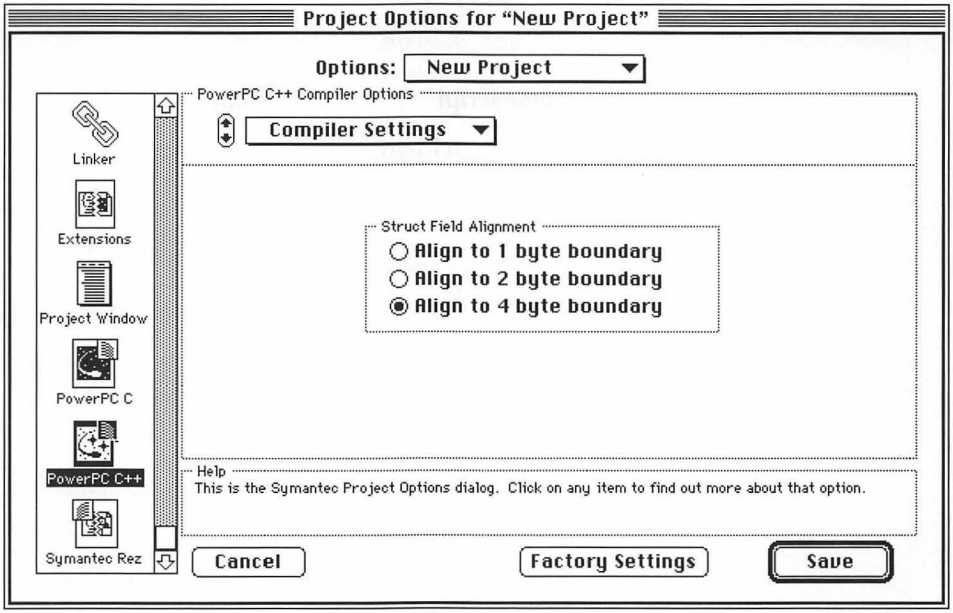

Figure 5-5 The Compiler Settings page

Struct field alignment

The padding options for the fields in structs, unions, and classes are as follows.

Align to 1 byte boundary. This option places all fields in structures, unions, and classes in memory without padding. (If not used with care, this option can result in odd-sized data structures.) The original setting is off.

Align to 2 byte boundary. With this option, all fields in structures and classes are padded out to word or 2-byte boundaries. The original setting is off. (Same as `#pragma [SC] options align=mac68k`)

Align to 4 byte boundary. This option pads out all fields in structures, unions, and classes to 4-byte (long-word) boundaries. This is the default setting. (Same as `#pragma [SC] options align=power`, or `#pragma [SC] options align=native`)

pragma option name

`struct_align`

AppleScript	Values

`struct_align`	`1`
	`2`
	`4`

Code optimization

The Code Optimization page lets you control how Symantec's PowerPC C and PowerPC C++ compilers optimize your code.

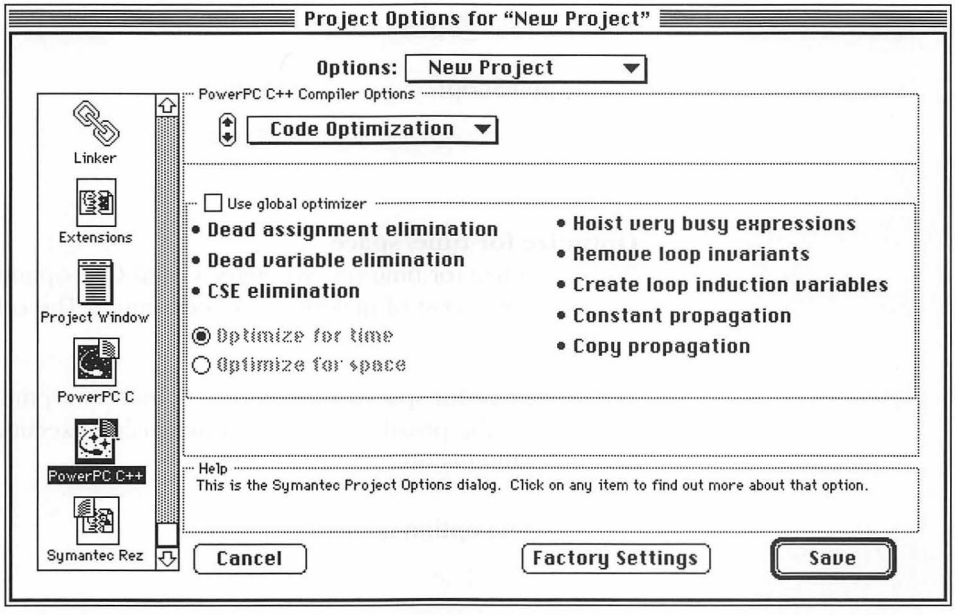

Figure 5-6 The Code Optimization page

Use global optimizer

This option controls the global optimizer. If it's off, the optimizer is not used. If it's on, you can turn on either Optimize for time or Optimize for space. Optimize for time results in faster code at the possible cost of code size. Optimize for space results in smaller code at the possible cost of execution time. The original setting is off.

You probably won't use the global optimizer while you're debugging. It adds a pass over your compiled code and may more than double your compilation time. Also, it generates machine code

that is much harder to manually map back to your source code. The debugger may not be able to pick out the machine code instructions that correspond to a given statement.

pragma option name

`global_optimizer`

AppleScript	Values
`global_optimizer`	`True` `False`

Optimize for time/space

With Optimize for time on, Symantec C and C++ optimize for speed at the possible cost of making your code larger. The original setting is on.

With Optimize for space on, Symantec C and C++ optimize to reduce code size at the possible expense of increasing execution time. The original setting is off.

pragma option name

`gopt_time`

AppleScript	Values
`gopt_time`	`optimize_time` `optimize_space`

Optimizations

The following are the optimizations that the global optimizer will perform.

Dead assignment elimination

This feature of optimization allows Symantec C and C++ to remove assignments to variables that are not used after being assigned, making your code smaller and faster to execute. This optimization also allows Symantec C and C++ to reuse registers for more than one variable.

With Dead assignment elimination, Symantec C and C++ do not load data into a register when that data is already in one. This optimization makes your code smaller and faster.

To understand how this optimization works, consider this example:

```
int j=0, i=1, k;

j = i + 1;
k = j;
```

When you reach the last statement (`k = j`), the value of `j` is found in two places: in a register and in memory. If this option is off, `k` gets the value from memory, requiring you to compute the memory address. If the option is on, `k` gets the value from the register, saving the time and space that computation takes.

Symantec C and C++ load `j` from memory into a register twice, once for each time it appears. With this feature, Symantec C and C++ load it from memory only once.

If you're debugging, you may want to turn off global optimization with its comprehensive set of features. When you set a variable in a data window, the debugger puts the new value into memory. Dead assignment elimination allows your program to use a value in a register and not in memory.

For example, look at the code above. After `j = i + 1` executes, there are two copies of `j`: one in memory and one in a register. Your program uses the register copy. The value in the register and in memory is `2`. If you examine `j`, the data window shows `2`.

Now, assume you change the value of `j` in the data window to 10. When your program continues, `k` is still set to 2. You changed the value of `j` in memory, but your program used the value of `j` in the register.

Dead variable elimination
This feature of global optimization allows Symantec C and C++ to determine the live ranges of variables in your code, removing any variables that have empty live ranges.

This optimization also allows Symantec C and C++ to reuse registers for more than one variable.

CSE elimination
This optimization makes your code smaller and faster. CSE (Common Subexpression) elimination replaces each subexpression that is used more than once with a temporary variable set to the subexpression's value. For example, consider this code:

```
a = i*2 + 3;
b = sqrt(i*2);
```

With this optimization, your code assigns `i*2` to a temporary variable and computes it only once. It's as if the code were written as:

```
temp = i*2;
a = temp + 3;
b = sqrt(temp);
```

Use this optimization on all your code.

Hoist very busy expressions
The compiler produces a single version of an expression that occurs over several different paths in the code. The result is smaller code.

Remove loop invariants
This optimization makes your loops faster. This feature moves expressions out of loops that remain constant in each iteration. For example, consider this loop:

```
while ( !feof(fp) ) {
   i = x*5;
   DoSomething( fp, i );
}
```

The compiler moves `i = x*5` outside this loop and computes it only once, as if you had written the loop like this:

```
i = x*5;
while ( !feof(fp) )
   DoSomething( fp, i );
```

Use this optimization if your code has many loops.

Create loop induction variables

This optimization makes loops faster, especially those that cycle through an array. For example, consider this loop:

```
int a[ARRAY_SIZE], i;

for (i=0; i<ARRAY_SIZE; i++)
   a[i] = GetNextElelment();
```

Without the create loop induction variables feature, the compiler performs a multiplication each time it figures the address for the next array element (`i * sizeof(int)`). With this optimization, the compiler remembers the address of the last element and adds the size of an element to that address. This optimization is beneficial if your code has a lot of loops. Note, however, that it may make your code slightly larger.

Constant propagation

Constant propagation replaces certain variables with constants. Consider the code:

```
A=5;
for(i=0; i<A; i++)
      abc[i]=A;
```

`A` always has the value `5` within the loop body. To optimize, the compiler replaces `A` with its value:

```
A=5;
for(i=0; i<5; i++)
      abc[i]=5;
```

Constant propagation opportunities occur frequently when loop rotation is done. For example, constant propagation converts:

```
while (e)
      expression;
```

to:

```
if (e)
   do
      expression;
while (e)
```

Copy propagation

Copy propagation is similar to constant propagation, except that it copies variables instead of constants. For example, it replaces:

```
A=b;
for(i=0; i<A; i++)
        abc[i]=A;
```

with:

```
A=b;
for(i=0; i<b; i++)
        abc[i]=b;
```

Copy propagation frequently uncovers unnecessary assignments, such as the assignment to `A`, which can be removed.

Debugging

The Debugging page lets you specify how the Symantec C and C++ compilers generate code for debugging. The debugging options are the same for Symantec C and C++ except Use function calls for inlines is only available with C++. Figure 5-7 shows the Debugging page for Symantec's PowerPC C++ compiler, Figure 5-8 shows the Debugging page for Symantec's PowerPC C compiler.

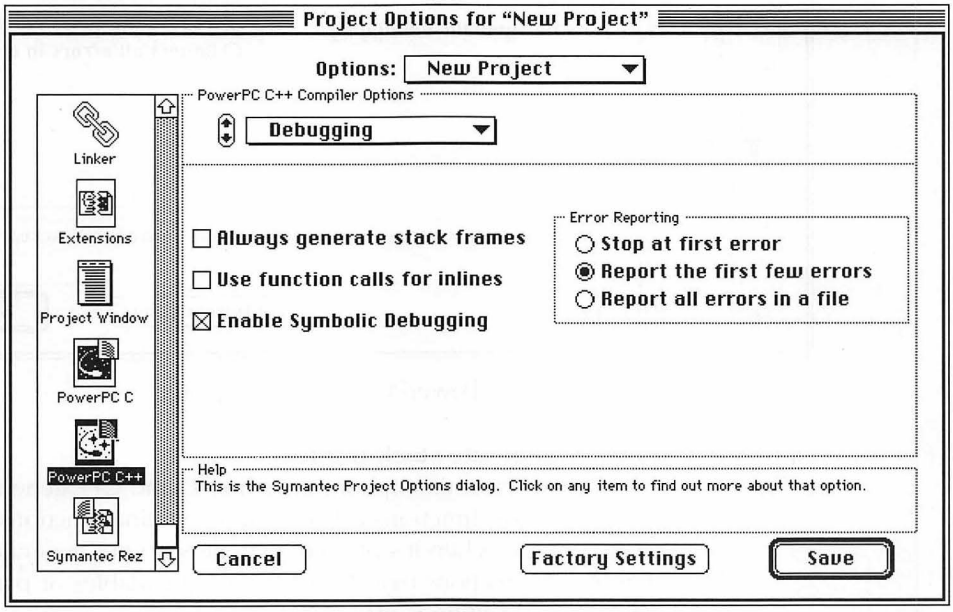

Figure 5-7 The PowerPC C++ Debugging page

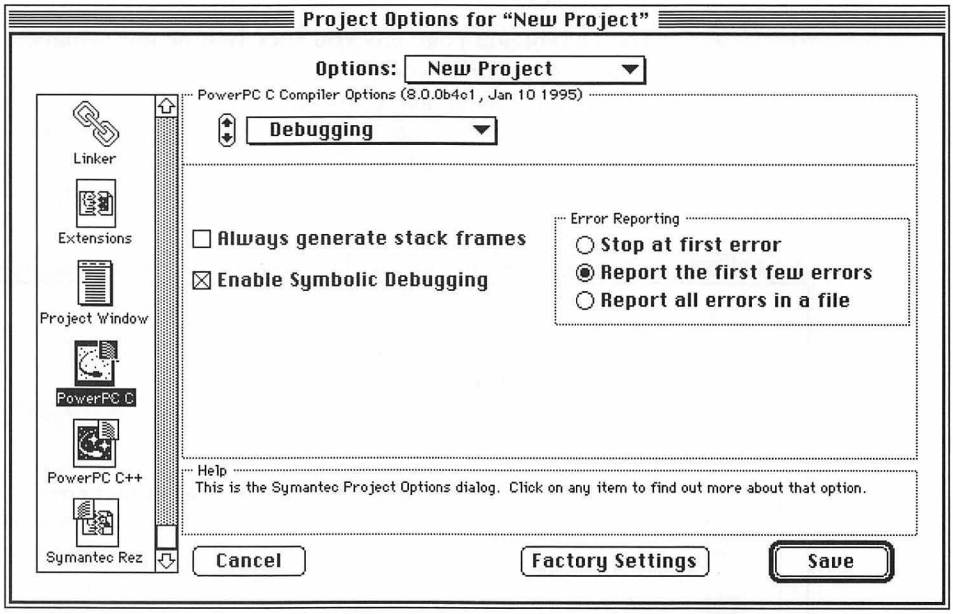

Figure 5-8 The PowerPC C Debugging page

Always generate stack frames

When you select this option, Symantec C and C++ generate a stack frame for most functions called, although inline functions don't have stack frames. When it's off, the compilers do not generate stack frames for functions that don't have local variables or parameters. The original setting is off.

If you're using the debugger, turn the option on. Otherwise, leave it off. Your program will be smaller and faster without the unnecessary stack frames.

pragma option name

```
force_frame
```

AppleScript	Values
`force_frame`	`True` `False`

Use function calls for inlines
(Symantec C++ only) When you check this option, Symantec C++ uses a function call for any inline functions. This allows easier debugging of inline functions. The original setting is off.

pragma option name

`dont_inline`

AppleScript	Values
`dont_inline`	`True` `False`

Enable Symbolic Debugging
When you select this option, Symantec C and C++ generate source-object correspondence, and symbolic variable information. When this option is off, the compilers will only generate source-object correspondence. Use this option to dramatically reduce the size overhead for projects where symbolic debugging of variables is not required. The original setting is on.

pragma option name

(none)

AppleScript	Values
`generate_symbolics`	`True` `False`

Error reporting
Each of these three options produces a different level of error reporting.

Stop at first error. If this option is on, Symantec C and C++ stop at the first error in your source file. The original setting is off.

Report the first few errors. If this option is on, Symantec C and C++ report the first few errors in your code, or stop at the first unrecoverable error. The original setting is on.

Report all errors in a file. If this option is on, Symantec C and C++ report errors found in your source file, or stop at the first unrecoverable error. The original setting is off.

Refer to Chapter 4, "Compiler Reference," for additional information about these pragmas.

pragma option names

`stop_at_first_err`

`report_all_err`

AppleScript	Values

`error_reporting`	`stop_at_first_err`
	`report_first_few_err`
	`report_all_err`

Warning Messages

Many of the Warning message options are the same for both Symantec C and C++. Figure 5-9 shows the Warning message page for Symantec's PowerPC C++ compiler, Figure 5-10 shows the Warning message page for Symantec's PowerPC C compiler.

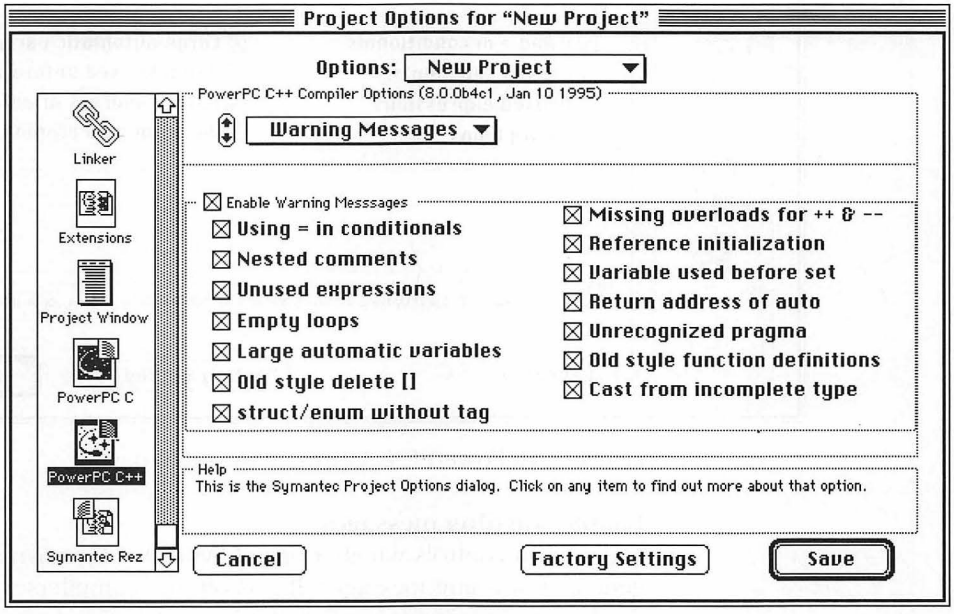

Figure 5-9 PowerPC C++ Warning Messages page

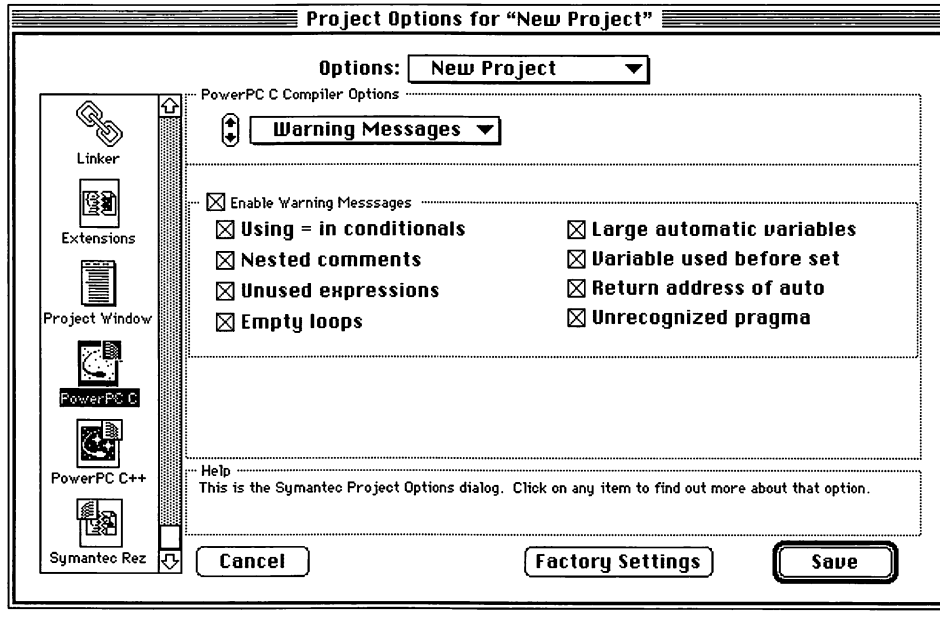

Figure 5-10 PowerPC C Warning Messages page

Enable warning messages

This option controls whether or not Symantec C or Symantec C++ generates warning messages. If it is off, the compilers will not display any warning messages. If it is on, you can choose options for the individual warning messages.

pragma option name

`generate_warn`

AppleScript	Values
`generate_warn`	`True` `False`

Choosing Factory Settings causes all warning message options to revert to their original settings. By default, the Enable Warning Messages option is set and each individual warning is enabled.

The following sections describe these messages and the conditions under which the compilers generate them. Cases where Warning message options are not available in the PowerPC C compiler are noted.

Using = in conditionals
If this option is on, Symantec C and C++ warn you when the conditional expression of a `for`, `if`, or `while` statement contains an assignment. The original setting is on. For example:

```
if (x = y) { ... }  // WARNING: possible
                    // unintended assignment
```

The warning points out that you may have meant this:

```
if (x == y) { . . . }
```

In cases in which the assignment is intentional, you can avoid the warning by rewriting the code, with identical results, as:

```
if ((x = y) != 0) { . . . }
```

pragma option name

`warn_unintended_assign`

AppleScript	Values
`warn_unintended_assign`	`True` `False`

Nested comments
If this option is on, Symantec C and C++ produce a warning when C style comments are nested. The original setting is on. For example:

```
/* This file contains the source code for the project
/* By: John Doe   */  // Warning: can't nest comments
```

pragma option name

`warn_nest_comments`

AppleScript	Values
`warn_nest_comments`	`True` `False`

Unused expressions
If this option is on, Symantec C and C++ alert you when the value of an expression has not been used. The original setting is on.

```
x == y;          // Warning: value of
                 // expression is
                 // not used
```

pragma option name

`warn_unused_expressions`

AppleScript	Values
`warn_unused_expressions`	`True` `False`

Empty loops
If you enable this option, Symantec C and C++ produce a warning when a semicolon appears immediately after an `if`, `while`, or `switch` statement. The original setting is on. For example:

```
if (x==y);   // Warning: possible
             // extraneous ';'

cout << "x==y" << endl;
```

If the semicolon is intentional, add white space after the end of the statement prior to the semicolon:

```
if  (x==y)
   ;
cout << "x may or may not equal y" << endl;
```

pragma option name

`warn_empty_loops`

AppleScript	Values
`warn_empty_loops`	`True` `False`

Large automatic variables
With this option, Symantec C and C++ warn you when the total size of automatic variables in a procedure is larger than 32KB. For example:

```
void f(void)
{
   int i[32000]
      // Warning: very large automatic
}
```

This code can cause a stack overflow. In such cases a dynamic memory allocation using `operator new` in C++ or `malloc` in C may be preferred. The original setting is on.

pragma option name

`warn_large_auto`

AppleScript	Values
`warn_large_auto`	`True` `False`

Old style delete [ ]
(Symantec C++ only) If this option is on, Symantec C++ warns you against using the older-style array `delete` operator. The original setting is on. For example:

```
delete [10] p; // Warning: use delete[]
               // rather than delete[expr],
               // expr ignored
```

pragma option name

`warn_old_style_delete`

AppleScript	Values
`warn_old_style_delete`	`True` `False`

struct/enum without tag
(Symantec C++ only) If this option is on, PowerPC C++ produces a warning when structs that are not given tags are used in the signature of a function or template. For example:

```
typedef struct {
...
}  *PX;
void f(PX *x)
{
}
// Warning: no tag name for struct or
// enum appearing in signature for 'f'
```

pragma option name

`warn_struct_without_tag`

AppleScript	Values
`warn_struct_without_tag`	`True` `False`

Missing overloads for ++ & --
(Symantec C++ only) This option produces a warning when you use the postfix versions of the `++` or `--` operators instead of the missing corresponding prefix operators, or the prefix version of the `++` and `--` operators instead of the missing corresponding postfix operators. The original setting is on. When ANSI conformance is turned on, this warning becomes an error and cannot be disabled. For example:

```
A& operator ++();
a++;              // WARNING: using
                  // operator++() (or --)
                  // instead of missing
                  // operator++(int)
```

pragma option name

`warn_missing_overload`

AppleScript	Values
`warn_missing_overload`	`True` `False`

Reference initialization
(Symantec C++ only) If this option is enabled, Symantec C++ produces a warning when a reference is initialized with a temporary value. The original setting is on. If ANSI conformance is turned on, this warning becomes an error and cannot be disabled. For example:

```
void f(int &);

f(2);           // WARNING: non-const
                // reference initialized to
                // temporary
```

pragma option name

```
warn_ref_init
```

AppleScript	Values
warn_ref_init	True False

Variable used before set
In Symantec C and C++, this option warns you when an attempt is made to obtain the value of an uninitialized variable. The original setting is on. This error is detected for the last line in the function in which it appears; use the name of the variable appearing in the warning message to determine where the error appears. This problem can be detected only when the global optimizer is enabled. For example:

```
void f(int);
void g() {
   int a;
   f(a);
}               // WARNING: variable 'a'
                // used before set
```

pragma option name

```
warn_used_before_set
```

AppleScript	Values
warn_used_before_set	True False

Return address of auto

If you enable this option, Symantec C and C++ produce a warning when the address of an automatic variable is the return value from a function. The original setting is on. For example:

```
int *f(void)
{

   int a;
   return(&a); // WARNING: returning address
               // of automatic 'a'
}
```

pragma option name

`warn_return_addr_auto`

AppleScript	Values
`warn_return_addr_auto`	`True` `False`

Unrecognized pragma

With this option, Symantec C and C++ produce a warning when they do not recognize a `#pragma` directive. The original setting is on. If `SC` appears immediately after the `#pragma` directive, this warning becomes an error and cannot be disabled. For example:

```
#pragma nooptimize(g)
               // WARNING: unrecognized
               // pragma
```

pragma option name

`warn_unrecognized_pragma`

AppleScript	Values
`warn_unrecognized_pragma`	`True` `False`

Old style function definitions
(Symantec C++ only) In Symantec C++, pre-ANSI function definitions are not allowed. When you use strict ANSI conformance, this produces an error and cannot be disabled. For example:

```
int f(x);
double x;
{
}
```

pragma option name

```
warn_old_style_definition
```

AppleScript	Values
warn_old_style_definition	True False

Cast from incomplete type
(Symantec C++ only) This warning is issued when a pointer or reference to an incomplete structure type is cast to a pointer or reference to another structure type. This is a warning because if the incomplete type is a sub-class of the type, incorrect code will be generated. For example:

```
struct X *px;
struct Y *py;
void f(void)
{
   px = (X*)py;       //warning
}
```

This warning can be avoided by declaring the struct or class that is being cast from. This usually means inclusion of the header containing the declaration.

pragma option name

```
warn_cast_incomplete_type
```

AppleScript	Values
warn_cast_incomplete_type	True False

Prefix

The Prefix page lets you write code that Symantec C and C++ will include in all your files.

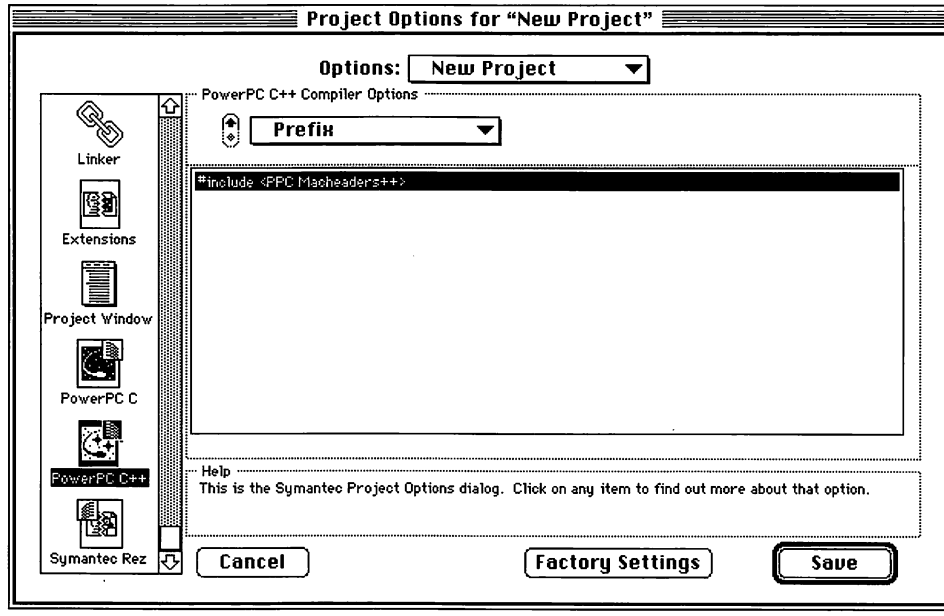

Figure 5-11 The Prefix page

Use the Prefix page to automatically include the same text in all the C or C++ source files for a project. The effect is the same as if you manually put the code into the files.

In PowerPC C++, if you use a precompiled header file such as `PPC MacHeaders++`, include it here. By default, this page contains the line: `#include <PPC MacHeaders++>`.

In PowerPC C, if you use a precompiled header file such as `PPC MacHeaders`, include it here. By default, this page contains the line: `#include <PPC MacHeaders>`.

If you need to define a macro in all your files, define it here. For example, you may have some debugging code in your files that's compiled only if the macro `DEBUG` is defined. To include that code, include this line here:

```
#define DEBUG
```

When you don't need to include the debugging code anymore, delete that line from this page. You don't need to edit every C or C++ source file in your project.

pragma option name

(none)

AppleScript	Values
prefix	*text string*

Porting Code 6

This chapter helps you port existing code to Symantec C++ for Power Macintosh. It also describes the PowerPC calling conventions, and how registers and the stack frame are used to pass parameters.

Contents

Porting from 68K

The recommended way to port code to the PowerPC is to begin by making the code as portable as possible. Once the code is portable, you then convert the items that are specific to the application being ported. The following list is tailored for porting 68K applications from THINK C/Symantec C++, but most of these items apply to all of the 68K development environments.

Porting steps performed on the 68K machine

You can make your code as portable as possible from within your present 68K development environment. Perform the following on your 68K Macintosh.

Upgrade to the latest headers

Apple recently changed their system headers to make them more consistent between platforms and between languages. The new headers, called the "universal headers" by Apple, were available with the 7.0 release of THINK C/Symantec C++ for the Macintosh, and are included with this version. Certain definitions, most notably patterns, have undergone some changes that require source code modifications. Make sure you are using these headers on the 68K prior to beginning the move to PowerPC. PowerPC development requires these headers.

Note

The correct way to determine if the universal headers are being used is to check for `__CONDITIONALMACROS__` being defined. If it is defined, the new headers are being used.

In the following example, the `GetIndPattern` call will fail to compile without the universal headers:

```
void myDraw(void)
{

    Pattern         thePat;
    Rect            theRect;

    GetIndPattern( thePat, sysPatListID,
                   12 ); /* system pattern */

    ...
}
```

In order to make it compile with the new definition of `GetIndPattern`, you would write:

```
GetIndPattern( &thePat, sysPatListID,
               12 ); /* system pattern */
```

Rework inline assembly

Any inline assembly must be rewritten in C or C++, or ported to PowerPC assembler in separate `.asm` files. Inline assembler is not supported in this release of Symantec C++ for Power Macintosh. When porting from 68K to PowerPC, remember the areas of your application that are critical to performance tend to change due to the differences in architecture. For example, floating-point operations increase in performance disproportionately from other portions of an application. For this reason, it is recommended that you move inline assembler into C or C++ until you determine these routines are as performance critical on the PowerPC as they were on 68K.

Remove int and structure size assumptions

On the PowerPC, the most efficient data types are `int` and `double`. All integer operations on the PowerPC are done on 32 bit registers, and all floating-point operations are done using 64 bit IEEE floating-point. Any manipulation of objects of other sizes requires additional work for the compiler to guarantee that the sign of the operand is correct. Also, the 32 bit bus of the PowerPC makes it more efficient to manipulate objects 32 bits at a time. For these reasons, it is recommended that you prefer the `int` and `double` data types for code being ported to PowerPC. Structures should be aligned on 4-byte boundaries (`#pragma options align=powerpc`) by default; this makes them more efficient for access purposes.

Note

Due to backward compatibility with the toolbox, many of the toolbox routines use `mac68k` alignment. If your application contains structures that are not used to interface with the toolbox and are not written to disk in binary form, it is more efficient to align them for the PowerPC.

For example, accesses to the following structure will suffer a performance penalty if it is not aligned on a 4-byte boundary:

```
#pragma options align=mac68k
struct X {
   short    s;
   int      i;
} ;
#pragma options align=reset
```

On 68K machines, this structure is 6 bytes and the `int` starts at offset 2 in the structure. On the PowerPC, it is more efficient to use 4-byte structure alignment, which makes the structure size 8 and `i` is at offset 4. Whenever possible, use the default alignment or explicitly set the alignment to `powerpc` with `#pragma options align=powerpc` prior to a `struct` declaration.

Use function prototypes

On the PowerPC, it is very important for all functions to be properly prototyped, and it makes the code easier to read and less prone to unexpected errors. If your C code does not already contain proper function prototypes, it would be a good idea to add them prior to converting to the PowerPC. For example, the following code will work on 68K (with 2-byte `int`s), and will not work on the PowerPC:

```
void f( x,  y)
short       x;
short       y;
{
      printf("%d, %d\ n", x, y );

}
void main(void)
{

      f(0x00100020L); /* x = 0x0010 and
                         y = 0x0020 */
}
```

This example illustrates one of the potential pitfalls involved in porting code from 68K to the PowerPC. It is equally important to explicitly declare the return type for each function, as this can also lead to incompatibilities.

Use strict pointer type checking
Enable strict pointer type checking, and cast as needed. This is important because without strict type checking, it is easy for assumptions regarding the sizes of data types to sneak in. For example:

```
void assign_x_y( short *x, int * y)
{
   while (*x++ = *y++)
   ;
}
```

This code assumes that `short` and `int` are the same size, which is not true on the PowerPC or in our 68K C++ compiler.

Access low memory globals properly
Access to low memory globals should be made through `LMGetxxx` and `LMSetxxx`. This can be done on the 68K as these interfaces exist in the 68K universal headers.

For example, the following code uses the current working directory low memory global defined in `<lomem.h>`:

```
#include <lomem.h>

long Check_working_directory(void)
{
   long cwd =  CurDirStore ;

   ...
}
```

The code should be changed to use the equivalent `LMGetCurDirStore()` function as in:

```
#include <LowMem.h>

long Check_working_directory(void)
{
   long cwd =  LMGetCurDirStore() ;

   ...
}
```

This code will now work on both PowerPC and 68K machines.

Note

A ramification of this suggestion is that you should no longer use the <lomem.h> header file. These addresses are not the same on PowerPC.

Make all code 32-bit clean

It is always best not to assume that you can use any bits in an address; moving to the PowerPC is just another source of problems for code that is not 32-bit clean.

Avoid direct access to registers and VIAs

Direct access to any hardware registers or VIAs should be avoided; it can be a source of problems between different 68K machines. If a system interface exists for the register or VIA, it should be used in preference to direct access.

Use 8-byte IEEE floating-point format

Use the 8-byte doubles option of THINK C and Symantec C++, and remove references to `long double` in your program. The 10- and 12-byte `double` formats supported on the 68K are not supported on the PowerPC. The highest available precision of the hardware floating-point unit is 8-byte IEEE format. When porting 68K code, make sure to eliminate dependency on 10-byte and 12-byte `double` formats for accuracy. See *Inside Macintosh PowerPC Numerics Manual* (Addison-Wesley) by Apple Computer for more details on the available floating-point support on the PowerPC. Following these suggestions will ensure that the 68K and PowerPC versions of your software will retain identical behavior.

Verify #pragma, #ifdef, and #ifndef statements

Examine all `#pragma`, `#ifdef`, and `#ifndef` statements to be sure they are meaningful on your compiler. A detailed list of the predefined macros and available pragmas can be found in "Compiler Reference," Chapter 4.

Note

On the PowerPC, `THINK_C` and `THINK_CPLUS` are no longer predefined symbols. Any code that is conditionalized on these should be examined, and if appropriate, converted to use the `SYMANTEC_C` and `SYMANTEC_CPLUS` predefined symbols.

Porting steps performed on the Power Macintosh

After you address the previous items, your code is as portable as 68K code can be. The next step is to convert those items that are specific to an application being ported to the PowerPC. These conversions are best performed using the native Symantec Project Manager because they are primarily PowerPC specific, however, they can be done in the 68K environment as well. The end result is a very portable application that has very few dependencies on either platform.

Convert callbacks to universal procedure pointers

The single largest change you will make to your code is to change all uses of callback routines into universal procedure pointers. This is necessary because the Macintosh operating system did not want to make assumptions regarding whether callback routines would be written in 68K or PowerPC code. A universal procedure pointer is callable from the operating system as either a PowerPC or a 68K procedure. For example, if you currently have a callback for performing custom control tracking, you might have code that looks like this:

```
#include <Controls.h>

pascal void myAction(ControlHandle
                theControl, short ctlPart);

int myTrack(ControlHandle theControl,
                Point localPt )
{
   return( TrackControl( theControl,
                localPt, (ProcPtr)myAction ));
}
```

This code will not compile on the PowerPC because the `TrackControl` function expects a `UniversalProcPtr` as its argument and not a `ProcPtr`. The code for myTrack would be converted as:

```
int myTrack(ControlHandle theControl,
                Point localPt )
{
   UniversalProcPtr myActionProc;
   myActionProc = NewRoutineDescriptor(
                (ProcPtr)myAction,
                uppControlActionProcInfo,
                GetCurrentISA());
   return( TrackControl( theControl,
                localPt, myActionProc ));
}
```

With this definition, the code will compile and run on either the 68K or the PowerPC. The `GetCurrentISA()` call returns a constant indicating whether 68K code or PowerPC code is being executed. `uppControlActionProcInfo` is a system-defined constant that should be available for all of the system callback routines describing their parameter lists for the `NewRoutineDescriptor` function. See *Inside Macintosh, PowerPC System Software* (Addison-Wesley) by Apple Computer for a detailed discussion of universal procedure pointers.

Modify performance-critical code resources

Code resources such as CDEF and MDEF resources are not required to be modified unless they are performance-critical. We recommend leaving them as 68K emulated code unless you specifically have a performance problem with a particular code resource. If you do need to write these as native code, see *Inside Macintosh, PowerPC System Software* (Addison-Wesley) for more information on creating native code resources.

Rewrite certain code resources as code fragments

Other code resources used by the application and not specifically called by the operating system should be rewritten as code fragments, if performance is an issue. For example, if your code used code resources for some self-modifying behavior or for user-extensions, modify these to use separate code fragment files that are loaded specifically using the `GetDiskFragment()` call. For more information on using code fragments see "Using the Standard Libraries," Chapter 7, and *Inside Macintosh, PowerPC System Software* (Addison-Wesley).

Porting from MPW C++

Wherever possible, Symantec C++ has striven for compatibility with the 68K MPW C++ compiler and the PowerPC MPW compilers, MRC and PPCC.

Note

MRC uses the Symantec front-end, so source level compatibility is extremely high.

Symantec C++ uses the same object format as most other MPW compilers from Apple, Lucid, IBM, and others. This allows you to import object code directly from these MPW compilers. Symantec C++ may use different mangling conventions for C++, and recompilation may be necessary. The following are other known differences between the compilers:

Include file search path

In Symantec C++, include directives in the form:

```
#include <filename>
```

search only the compiler include directories for the file. They do not search the user directories.

enum prototyping

Symantec C++ defaults to the Macintosh convention of sizing an `enum` to the smallest data size (`char`, `short`, or `int`) that holds the `enum` range. MPW C++ does the same, but other implementations handle `enum` prototyping differently. Check your compiler documentation and set the Enums are always int setting to match the behavior of your own compiler. For additional information on this option, see Chapter 5.

Structure definition

Both the Symantec and MPW compilers place fields in structures, but they align bit-fields differently. Symantec C++ packs bit-fields according to the size of the type of the containing field.

For example:

```
struct a
{
    char x : 2;
    char y : 2;
    short z : 2;
    int zz : 15;
};
```

Symantec compilers start field `z` on the next new short word while MPW compilers place field `z` in the same byte as field `x` and `y`. This difference can also result in a size difference in the structures. Symantec C++ allocates an integer (4 bytes) for field `zz`. MPW C++ allocates only 2 bytes.

Static member functions

You cannot declare static member functions as `const`. MPW C++ ignores the declaration. The Symantec C++ compiler gives an error message.

For example, the following statement is flagged as illegal by the Symantec C++ compiler:

```
static void DoSomeStuff() const;
```

const violations

Symantec C++ is stricter than MPW C++ regarding `const` declarations. The MPW compiler allows you to define a `const` member function that violates the `const` declaration; Symantec C++ refuses to compile incorrect function definitions. To port MPW C++ code, either rewrite your functions or don't declare functions as `const`.

Data definitions in precompiled headers

Symantec C++ does not support data definitions in precompiled headers. You may, for example, forget to declare a function inline when it is defined in a header file.

Instantiating abstract base classes

When you provide definitions for the pure virtual functions of an abstract base class, you must be careful to use the same function prototypes as were used in the virtual function declaration. For example:

```
struct X {
   // f() is a pure virtual function
   virtual void f(void *) = 0;
};

struct Y : X {
   virtual void f(const void *) { }
};
```

In this example, the member function `Y::f()` does not provide an implementation for `X::f()`. Symantec C++ correctly interprets `Y::f()` as an overloaded function based on `X::f()`. MPW C++ incorrectly interprets `Y::f()` as the pure virtual function's implementation.

PowerPC Calling Conventions

For additional information on the PowerPC Calling Conventions, see Inside Macintosh, PowerPC System Software *(Addison-Wesley) by Apple Computer.*

Unlike the 68K environment, the PowerPC uses one standard calling convention that routines use to call other routines. In summary, the PowerPC calling convention:

- Passes most parameters in dedicated registers, and only passes parameters on the stack when the available registers are exhausted.
- Determines stack frame size at compile time.
- Reserves specific stack frame sections for saved registers, parameters, local variables, and linkage information required for the stack frame.

Parameter passing

The PowerPC's large quantity of dedicated registers for parameter passing increase the likelihood that your subroutine's parameters will be passed in registers, reducing memory accesses, and consequently, increasing the performance of your application. In cases where registers are not used for parameter passing, it is necessary to have an understanding of how to use the parameter area of the stack frame.

Assigning parameters

The Symantec C++ for Power Macintosh compilers assign parameters to registers and to the parameter area in the caller's stack frame by the following procedure:

- Parameters are arranged in order, like the fields of a record. The first field is the leftmost parameter, the fields are aligned on 32-bit word boundaries, and any integer parameters that occupy less than 32 bits are extended to 32 bits.

Note

Even when these parameters are in registers, it is assumed that the callee must extend the sign of any integral parameters that are smaller than 4 bytes.

- When parameters are passed in registers, the first eight words are passed in GPR3 - GPR10. The first thirteen floating-point parameters are passed in FPR1 - FPR13.
- GPR3 and FPR1 are used to return simple function results.
- Custom data structures (C structures or Pascal records) are passed without expanding their fields for word alignment purposes. When this data is returned, the caller maintains adequate space for the result on the stack, places the result's address in GPR3, and starts placing the parameters in GPR4.
- If there are parameters that do not fit in the available registers, they are passed in the parameter area of the caller's stack frame.

Note

This convention should be followed by all PowerPC compilers.

Stack frame layout

The Symantec C++ for Power Macintosh compilers create an area in the caller's stack frame for the parameters that is big enough to contain all the parameters passed to the callee. The number of parameters that are actually passed in registers has no effect on the size of this area.

The following example illustrates how the compilers use the parameter area on the stack.

```
void  My_Func  (float f, int i, float f2,
               double d, char c, short s,
               unsigned u, float f3, long l);
```

Viewing the parameter list from My_Func as a structure gives:

```
struct   parameters  {
   float                f;
   int                  i;
   float                f2;
   double               d;
   char                 c;
   short                s;
   unsigned             u;
   float                f3;
   long                 l;
};
```

Although some of the variables in this example are passed in registers, the compiler will allocate space on the stack for all of them. In allocating space, the PowerPC uses the following lengths for variables: all integers, regardless of size, are 32 bits, floating-points are 32 bits, and doubles are 64 bits.

Figure 6-1 shows the parameter area on the stack for the `My_Func` example.

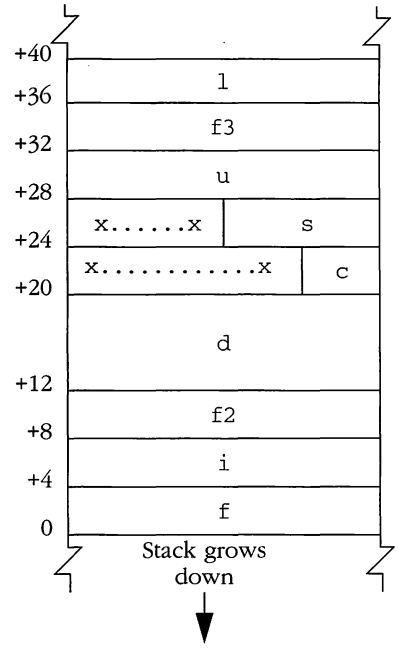

Figure 6-1 Parameter area on the stack

The stack starts with three 32-bit words for the first three parameters: a floating-point field, `f`, an integer, `i`, and another float, `f2`. The 64-bit `double`, d, is assigned to the next two words, followed by `c`, a `char`, and `s`, a `short` integer. Both the `char` and the `short` integer are placed into their respective words as follows: their values are placed into the lower half of their respective words, and the upper half of their words are padded. The sequence of placement continues for the remaining fields.

The relationship between the stack frame and registers

To understand which parameters are passed in registers and which are passed on the stack, it is necessary to map the stack to the available registers. Figure 6-2 illustrates which parameters in this example are passed in registers and which are passed on the stack.

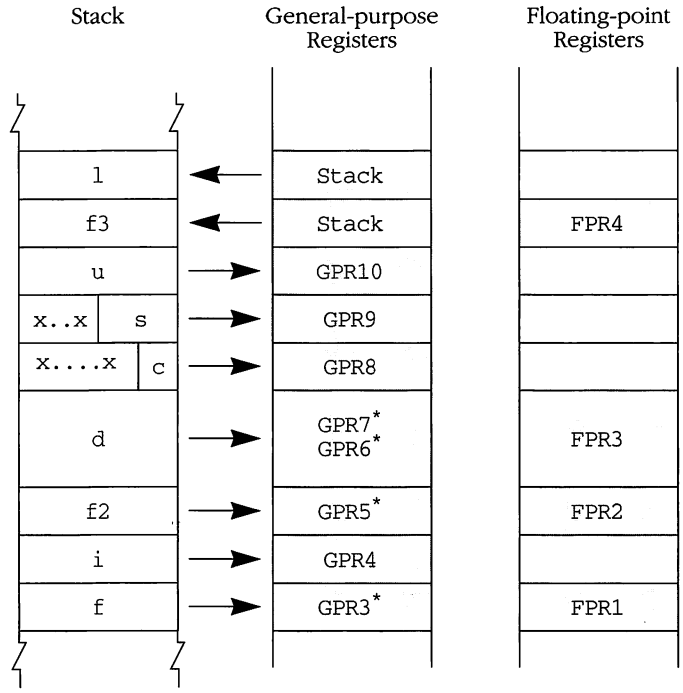

Figure 6-2 Parameters mapped to registers

* These registers are reserved, but the corresponding values from the floating-point registers are not stored in them.

Since `GPR3 - GPR10` and `FPR1 - FPR13` are the only registers that can be used for parameter passing, the stack must be used when these are no longer available.

Whenever a floating-point value parameter is placed in a floating-point register, a copy is placed in a general-purpose register. In the case of a `double`, one floating-point register is used and two

general-purpose registers must be used. These are always reserved, but used only when no prototype or a `...` prototype is used. For example,

```
int    printf(char *, ...);
          .
          .
          .
       printf("hello%f", 3.0);
```

The process continues until there are no registers available for the parameters. In the example in Figure 6-2, `u`, the `unsigned` integer, occupies the last of the general-purpose registers. This forces the next parameter, `f3`, a `float`, to be placed in the next floating-point register, and, since there are no available general-purpose registers for the copy, the parameter is passed in the register and on the stack. For more information about how the registers and stack are used, see *Inside Macintosh, PowerPC System Software* (Addison-Wesley) by Apple Computer.

Using the Standard Libraries

7

This chapter describes the standard libraries that come with Symantec C++ for Power Macintosh, how to use them with your project, and how to modify them for your own purposes. It also explains how to use the online reference to look up functions.

Contents

Headers and Libraries

The layout of the headers and libraries folders is a little different in Symantec C++ for Power Macintosh than in previous releases of Symantec C++.

Apple vs. Symantec standard libraries

For the PowerPC, there are two choices for the ANSI C standard libraries: the Apple standard libraries and the Symantec standard libraries. The Apple standard libraries are built into the PowerPC ROM, and your application will require less memory to use them. Source code for these libraries is not provided. The Apple libraries support building native MPW tools from within the Symantec Project Manager.

The Symantec standard libraries are included with full source code available. They are packaged as either statically linked or shared libraries. Use these libraries when customization of the libraries is required. Since performance is better when using statically linked libraries on the PowerPC, these libraries are a good choice for high performance applications.

Standard libraries

This folder contains the standard libraries for both the PowerPC and 68K environments. It also contains the following folders:

Folder	Contains
`Headers and Source`	The standard includes and the source code for the Symantec standard libraries. It also contains the project files used to build the PowerPC libraries.
`oops Libraries`	68K libraries used for programming with THINK C + Objects or Symantec C++ with Pascal Objects.
`STL`	The Hewlett-Packard Standard Template Library.
`Apple PPC Libraries`	Apple's standard libraries (binary only).

Table 7-1 outlines the library mapping from 68K to PowerPC when using the Symantec libraries.

68K Library	PPC Static Library	PPC Shared Library (requires...)	PPC Sub-project†
ANSI ANSI++	PPCANSI.o	PPCANSI	PPCANSI.π
ANSI-small ANSI-small++	PPCANSI_small.o	PPCANSI_small	PPCANSI_small.π
unix unix++	PPCunix.o	PPCunix (PPCANSI)	PPCunix.π
profile profile++	Profiling not supported in this release of Symantec C++ for Power Macintosh		
IOStreams	PPCIOStreams.o	PPCIOStreams (PPCANSI, PPCCPlusLib)	PPCIOStreams.π
complex	PPCcomplex.o	PPCcomplex (PPCIOStreams)	PPCcomplex.π
CPlusLib	PPCCPlusLib.o	PPCCPlusLib (PPCANSI)	PPCCPlusLib.π
CPlusLib TCL	PPCCPlusLib TCL.o	PPCCPlusLib‡	PPCCPlusLib.π

Table 7-1 Library mappings from 68K to PowerPC

† Use this version of the library to debug changes to the standard libraries that you make.

‡ PPCCPlusLib TCL is not available because the PPCCPlusLib shared library can be used instead. Local definitions of operator, new(), and delete() in the TCL will be preferred to the global definitions. When building a TCL application, remember to use PPCCPlusLib TCL.o if required.

Choose shared libraries if application size is an issue. Performance will be sacrificed. Using these libraries means that you will need to include these standard libraries as components of your shipping product.

Table 7-2 describes the libraries containing static run-time code that virtually all applications built for the PowerPC will require, both C and C++ based.

Library	For Use With...	When Using...	To Create...
`PPCRuntime.o`	Internal linker	Symantec Libraries	Application
`ApplePPCRuntime.o`	Internal linker	Apple Libraries	Application
`shlbPPCRuntime.o`	Internal linker	Symantec Libraries	Code fragment
`shlbApplePPCRuntime.o`	Internal linker	Apple Libraries	Code fragment
`MPWPPCRuntime.o`	External linker	Symantec Libraries	Application or code fragment
`MPWApplePPCRuntime.o`	External linker	Apple Libraries	Application or code fragment

Table 7-2 Required libraries for PowerPC applications

For more information on the internal and external linkers, see the *Symantec C++ User's Guide and Reference.*

Using the Apple standard libraries

The header files provided with Symantec C++ for Power Macintosh are set up for either the Apple or Symantec standard libraries. By default, the headers for the PowerPC are the Symantec standard libraries. In order to use the Apple standard libraries, you must `#define _USE_APPLE_LIBRARIES_` to be `1` before including any of the standard library header files.

Note

If you are using a custom precompiled header that includes any of the standard library headers, you must add this `#define` to the precompiled header source file and re-precompile it.

The Apple standard libraries differ from the Symantec standard libraries in their treatment of ANSI escape sequences within string literals and character constants. The Symantec libraries expect `\n` to be `0xa` and `\r` to be `0xd`. The library maps these characters at run-time to the correct characters (`0xd` and `0xa`). The Apple standard libraries expect this mapping to be performed by the compiler when it encounters these characters within character constants and string literals.

The Map carriage returns option (described in Chapter 5, "Compiler Options Reference") controls whether the compiler performs this mapping. This option must be on when using the Apple standard libraries, and off when using the Symantec libraries. By default, this option is off.

Apple provides libraries for ANSI C in ROM and for C++ `new` and `delete` as static libraries. The Symantec Project Manager fully supports the ROM use of either or both of these sets of libraries. You may also choose to use the Apple ANSI C library and the Symantec C++ libraries.

In order to use the Apple C standard libraries, you will need to add:

`StdCRuntime.o`	low-level C runtime library
`StdCLib.xcoff`	interface to the standard C library in ROM

Choose either `PPCCRuntime.o` (Apple's low-level run-time) or the appropriate version of Symantec's `PPCRuntime.o` library. If you are using C++, we strongly recommend you use the Symantec versions of `PPCCPlusLib.o` and the run-time library.

Macintosh libraries

To better support both PowerPC native and 68K development, the Macintosh-specific libraries and headers have been slightly reorganized. The `Mac Libraries` and `Mac #includes` folders are now contained in the `Macintosh Libraries` folder. A new folder, `More System Interfaces`, has been added to the `Development` folder containing headers and libraries for new Apple System Software technology such as the Drag Manager, AOCE, AppleScript, ColorSync, and the Thread Manager. Drag required files, as needed, into the `Macintosh Libraries` folder. This folder is

taken from the latest E.T.O. CD and is used as is with minor modifications.

The `Thread Manager` folder contained two versions of `Threads.h`, one in the `Interfaces` folder and one in the `SnakesWithSemaphores` folder. The latter is out of date and has been deleted. Also, the correct `Threads.h` was modified to remove the ',' after the final enumeration constant in the gestaltSelectors enumeration.

The `QuickTime` folder contains an `Interfaces` folder that holds more recent revisions of the QuickTime headers than are contained in the standard `Universal Headers` folder. As such the folder has been renamed to `(Interfaces)`.

Table 7-3 outlines the library mapping from 68K to PowerPC.

68K Library	PPC Library
`AppleTalk`	`InterfaceLib.xcoff`
`CommToolbox`	See `CommToolbox` dev kit.
`Graf3D`	Not supported on PowerPC
`HyperXLib`	See `HyperXLib` dev kit.

`MacTraps`
`Old MacTraps`
`MacTraps2`
	`InterfacLib.xcoff`

`nAppleTalk`	`InterfaceLib.xcoff`
`OSL`	`ObjectSupportLib.xcoff`
`QuickTime`	`QuickTimeLib.xcoff`
`SANE`	`MathLib.xcoff`†

Table 7-3 Library mappings from 68K to PowerPC

† `MathLib.xcoff` is not a direct substitute for SANE. See "*Inside Macintosh: PowerPC Numerics*" for details. In general `MathLib.xcoff` will be required if you link with `InterfaceLib.xcoff`.

Symantec ANSI Libraries

Symantec C includes two standard function libraries, ANSI and unix, which you can find in the `Standard Libraries` folder:

- The PPCANSI library is a complete implementation of the ANSI C standard library. The standard gives all C programmers in all ANSI C environments a consistent set of functions, making programs easier to port. It includes functions to perform file and screen I/O, string-handling, math, and more.

- The PPCunix library contains several functions common to Unix implementations of C that are not part of the ANSI standard library. It includes functions to handle files, signals, and other Unix-specific features.

Use the PPCunix library to make porting software from Unix systems easier. When writing a new program, try to avoid the PPCunix library, and use the PPCANSI library instead. The descriptions of the Unix functions in the online *Standard Libraries Reference* tell you when there is an equivalent function in the PPCANSI library.

The PPCANSI library includes the console package, which lets you use simple Macintosh windows called consoles. Consoles let you port MS-DOS and Unix programs easily and help you get simple C programs running quickly. For more information, see "Using the Online Standard Libraries Reference" later in this chapter.

Symantec C++ includes four standard function libraries, PPCCPlusLib, PPCIOstreams, PPCcomplex, and the Standard Template Library (STL), which you can find in the `Standard Libraries` folder:

- The PPCCPlusLib library contains utility routines for the Symantec C++ compiler that handle the `new` and `delete` operators for objects and arrays.

- The PPCIOstreams library is a full implementation of the standard stream input/output library described in Bjarne Stroustrup's *The C++ Programming Language*. It provides flexible and extendable facilities for input and output.

- The PPCcomplex library allows you to perform arithmetic operations on complex numbers.

- The STL is a set of template classes and functions from Hewlett-Packard that are of general utility. This library is not provided in binary (`.o`) form for this reason. For information on the Standard Template Library, see the online document, *STL-Online Doc*. Updates and additional information on STL can be found on the Internet ftp site //butler.hpl.hp.com.

Special versions of the standard libraries

Table 7-4 describes the special versions of the standard libraries, which are designed to handle special circumstances. Note that these libraries leave out some functions; therefore, use one of these libraries only if your program meets the condition for which that library was designed:

Use this library...	If your program...
PPCANSI_small	Does not use consoles or floating-point functions
PPCCPlusLib TCL	Uses the THINK Class Library

Table 7-4 Special versions of the standard libraries

The PPCANSI_small library differs from the ANSI library. The PPCANSI_small library is smaller than ANSI, and projects that use it are smaller as well. It leaves out many things that are available in the ANSI library. Most importantly, it leaves out console functions and floating-point support. Also, the `%f`, `%g`, and `%e` conversions are not available for the formatted I/O functions, such as `printf()` and `scanf()`.

Customizing the PPCANSI library

You may want to create your own version of the PPCANSI library that, for example, includes floating-point functions and leaves out console functions. To create your own version of the PPCANSI library:

1. Choose the library from which you want to start. For example, if you want your library to exclude console and floating-point functions, start with PPCANSI_small.

2. In the Finder, create a copy of the project and give it a new name, like `PPCMyANSI`.

3. In the Symantec Project Manager, open your renamed project.
4. In the **PowerPC C options** pages of the **Project Options**, set up your options. Table 7-5 contains some common settings for the options.
5. Remove files you don't need. For example, if you don't need time functions, remove `time.c`.
6. Choose **Bring Up To Date** from the **Project** menu to compile your project.

For more information on the Link Errors window, see the Symantec C++ User's Guide and Reference.

If you remove files from the ANSI library, you may get link errors when you compile your project. The Link Errors window lists the functions that other functions in your project need. Search the files you removed, and add back to your project the files that contain the needed functions to your project.

Table 7-5 describes some common options settings.

To create a library that uses...	Set these options...
No console functions	In the Prefix section, define the symbol `_NOCONSOLE_`, and remove the files `console.c` and `command.c`.
No floating-point functions	In the Prefix section, define the symbol `_NOFLOATING_`.

Table 7-5 Common option settings for the PPCANSI library

Using the Online Standard Libraries Reference

Symantec C++ for Power Macintosh includes several useful tools that make your work with the standard libraries much easier. An online version of the Standard Template Library (STL) documentation can be found in the online documentation. This document provides a comprehensive description of the STL.

The Symantec Project Manager also comes with an online *Standard Libraries Reference*, THINK Reference, that includes a database for the Symantec C standard libraries. In addition, another database that

describes the Symantec C++ class libraries PPCIOStreams and PPCComplex is included with Symantec C++ for Power Macintosh.

Note

THINK Reference refers to the libraries by their 68K names. The contents of the PowerPC versions of the libraries are identical to the 68K versions.

The version of THINK Reference included with Symantec C++ for Power Macintosh also contains the entire "*Inside Macintosh*" volumes I - VI, complete documentation for the THINK Class Library, and Symantec C++ for Power Macintosh's error messages.

To look up a standard library function in THINK Reference when you're in the Symantec Project Manager, select the function name and choose **Find in Doc Server** from the **Search** menu. You can also go to the Finder, double-click the THINK Reference icon, and type the function's name. Usually after a few seconds you'll see the THINK Reference window shown in Figure 7-1.

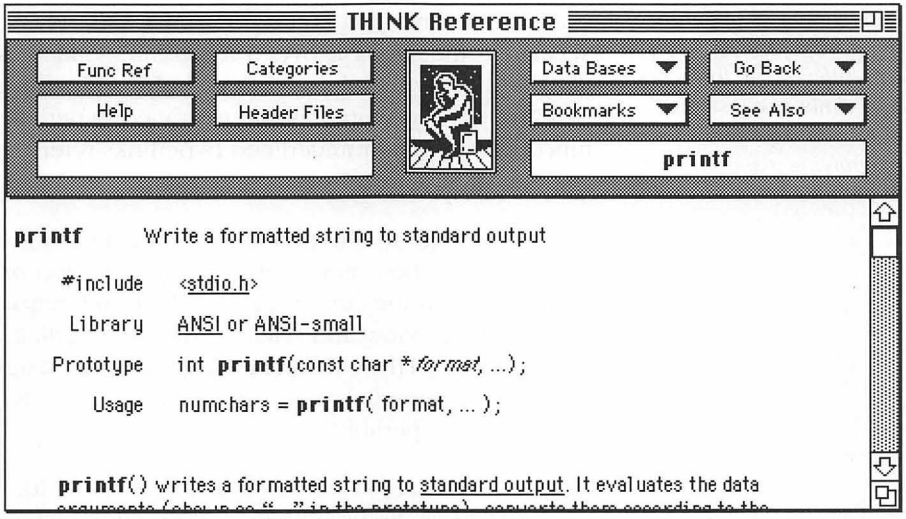

Figure 7-1 THINK Reference window

Note

If you moved the databases since you installed them, THINK Reference displays a dialog that asks you to locate them. See "Finding lost databases" later in this chapter for more information.

The top part of the THINK Reference window is the button panel, which contains buttons that help you move around the database. The box on the right side is the Page Title field and contains the name of the current field. The bottom part is the information area, which contains a page from the database.

Looking up a topic

If your Symantec C++ documentation refers you to a topic in the online *Standard Libraries Reference*, start THINK Reference, type the name of the topic, and press Return. THINK Reference brings you to the page that deals with that topic.

Moving around THINK Reference

Sometimes THINK Reference uses underlined words for emphasis. You can differentiate these from hyperlinks because the cursor changes to a magnifying glass only over hyperlinks.

You move around THINK Reference with hyperlinks. When you click a hyperlink, you go to a page that describes a related topic. Whenever the cursor is over a hyperlink, it changes to a magnifying glass (). Hyperlinks appear as **bold underlined** words or as plain underlined words. Bold underlined hyperlinks usually refer to functions, and plain underlined hyperlinks refer to types, variables, or fields of structures.

You can also go to a hyperlink by typing its name. What you type appears in the box on the left side of the button panel. If the hyperlink is on the current page, THINK Reference scrolls the page to bring it into view and selects it. If the hyperlink isn't on the current page, THINK Reference displays in the Page Title field the name of the hyperlink that matches what you've typed. Press Return to go to the hyperlink's page.

Read THINK Reference's online documentation for more information on the features discussed here and on other features such as searching, printing, and setting preferences. Just click the Help button in the button panel.

Reading a function reference page

Most function reference pages contain the four sections in Figure 7-2 — summary, description, returns, and example. Some may omit the returns and example sections.

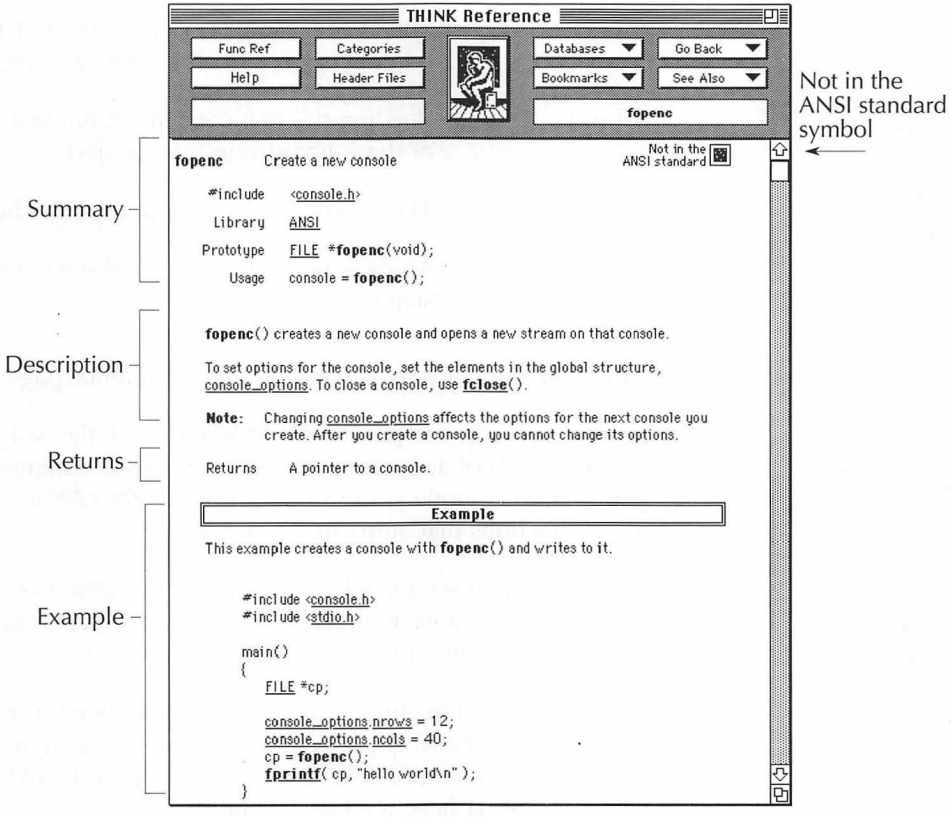

Figure 7-2 A function reference page

The summary section briefly describes the function in five lines, as described in Table 7-6.

This line...	Contains...
function name	The name of the function and a short description.
`#include`	The header file that declares the function. Include it in any source file that calls the function.
Library	The libraries that contain the function. Include one of these libraries in your project.
Prototype	The function's declaration from the header file.
Usage	An example of how to use the function in a statement.

Table 7-6 Summary section of a function reference page

If the function is not part of the ANSI standard, the Not in the ANSI standard symbol appears at the right side of the summary section. If you plan to compile your program with another ANSI C compiler, avoid functions that aren't in the standard.

The description section tells you what the function does and describes each of its arguments in detail. This section may contain notes that call attention to important information.

The returns section summarizes what the function returns. If the function sets `errno` when there's an error, this section also describes `errno`'s possible values. If a function doesn't return a value, it doesn't have a returns section.

Most functions also have an example section that illustrates how to use the function in a program. To copy the example to the clipboard, choose **Copy Code Examples** from the **Edit** menu. You can then paste it into a source file and try it out.

Using the tables of contents

The Quick-Jump buttons in the upper-left corner of the THINK Reference window bring you to tables of contents that help you look up the information you need.

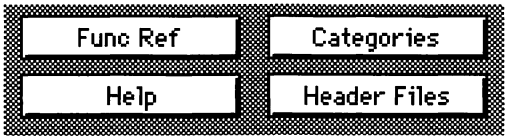

Figure 7-3 The Quick-Jump buttons

Which button you use depends on what you're looking for and how much you know before you start looking. For instance, if you want more information on a function and you know its name starts with `fget`, you would start in the Func Ref page. If you know that `stdio.h` has a function for printing error messages, the best place to start would be in the Header Files page.

When you're looking up something in a table of contents, remember that you can type the name of a hyperlink to select it. If the hyperlink doesn't appear in the window, THINK Reference scrolls the page until it appears. To go to the hyperlink's page, press Return.

Table 7-7 summarizes how you use the Quick-Jump buttons to look up information.

Use this...	**To look up...**
Func Ref	A function, if you know part of its name
Categories	A function, if you know roughly what it does
Help	Online help
Header Files	A data structure, type, or function, if you know its header file

Table 7-7 Uses of Quick-Jump buttons

Func Ref

The Func Ref page is an alphabetical listing of every function in the Symantec C standard libraries. To get to this page, you click the Func Ref button in the button panel, choose **Func Ref** from the **Reference** menu, or press Command-R.

Categories

The Categories page is a list of categories, such as I/O Functions, UNIX Functions, and Time Functions. This page is useful if you know what the function does but don't know its name. Each category has a page that contains hyperlinks to all the functions listed on it.

To get to the Categories page, you click the Categories button in the button panel, choose **Categories** from the **Reference** menu, or press Command-T.

Help

The Help page contains links to pages that explain how THINK Reference works. To get to the Help page, you click the Help button in the button panel, choose **Help** from the **Reference** menu, or press Command-N.

Header Files

The Header Files page contains an alphabetical list of hyperlinks to the header files in the Symantec C standard libraries. Each hyperlink takes you to a page that describes the types, macros, global variables, and functions that the header file defines.

To get to the Header Files page, you click the Header Files button in the button panel, choose **Header Files** from the **Reference** menu, or press Command-H.

Finding lost databases

If you have moved a database since you installed it, THINK Reference asks you to locate it. THINK Reference expects to find its databases in the folder that contains the THINK Reference application. THINK Reference displays the dialog shown in Figure 7-4. Once the name of the folder containing the database is displayed in the button under the prompt Use all database files in,

click the button. THINK Reference remembers the location of the database.

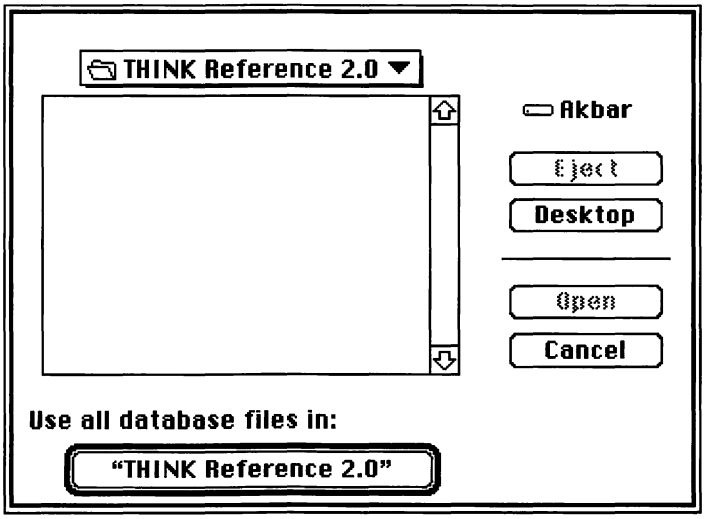

Figure 7-4 Selecting the database folder

Whenever you move a database, THINK Reference brings up this dialog the next time you use it. If you want THINK Reference to bring up this dialog, hold down the Shift key as THINK Reference is starting up. You may need this dialog if you have more than one set of databases and want to switch among them.

Using Symantec Rez 8

Resource description files are the text files that Symantec Rez compiles to produce resources for your application.

If you are not familiar with Apple's Resource description language, Rez, we recommend that you use a graphical resource editor, such as ResEdit (included in your package) or Resorcerer. See Apple Computer's *ResEdit 2.1 Reference* (Addison-Wesley) for instructions on using ResEdit.

This chapter describes how to use Symantec Rez to compile resource description files and add resources to your projects. It describes how to write a resource description file, how to set your Symantec Rez options for your project, and also how Symantec Rez is different from Rez, the resource compiler that comes with Apple's Macintosh Programmer's Workshop.

Contents

The Resource Compiler

The resource compiler, Symantec Rez, compiles a text file (or files) called a resource description file and adds the resources to your project resource file. Symantec Rez has preprocessor directives that let you substitute macros, include other files, and use if-then-else constructs. (These directives are described in the section "Preprocessor Directives" later in this chapter.)

Using a resource compiler

Symantec Rez creates new resources in your project resource file. Use it with the Resource Copier, which copies resources that already exist into your project resource file, to include resources to your applications. Figure 8-1 illustrates the process of creating a project resource file.

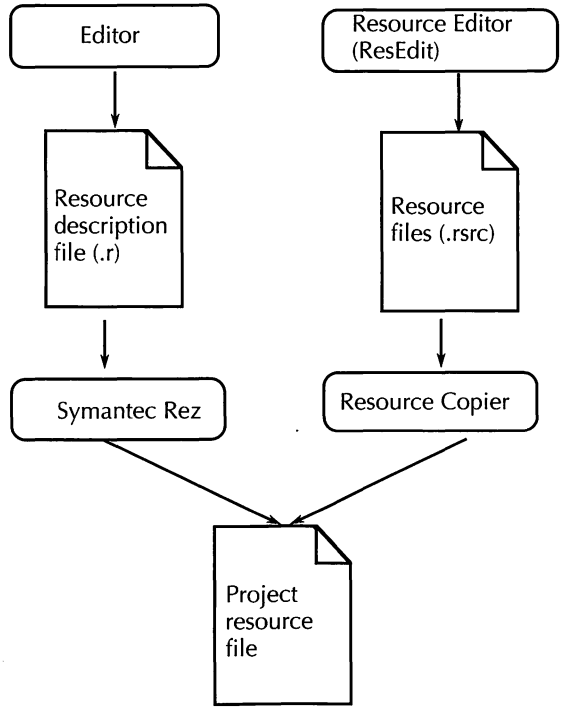

Figure 8-1 Creating a resource file

Standard type declaration files

`Types.r`, `SysTypes.r`, and `Pict.r` are examples of text files containing resource declarations for standard resource types. These

files and others are located in the Rez #includes folder, which is in the Macintosh Libraries folder in the Symantec C++ for Power Mac folder. Table 8-1 describes the standard type declaration files.

This file...	Contains...
Types.r	Type declarations for the most common Macintosh resource types ('ALRT', 'DITL', 'MENU', etc.)
SysTypes.r	Type declarations for 'vers', 'DRVR', 'FOND', 'FONT', 'FWID', 'intl', 'NFMT', and many other system-level resources.
BalloonTypes.r	Type declarations for Balloon Help resources ('hmnu', 'hdlg', 'hwin', etc.)
AEUserTermTypes.r AEWideUserTermTypes.r	Types for creating Apple Event terminology and resources ('aete', 'aeut', 'scsz')
CodeFragmentTypes.r	Type declaration for the 'cfrg' resource.
MixedMode.r	Type declarations for creating resource based Mixed Mode targets ('rdes', 'fdes', and 'sdes')
ImageCodec.r	Types for QuickTime image compression resources.
CTBTypes.r	Types for the Communications Toolbox.
InstallerTypes.r	Types for creating installer scripts for Apple's Installer.
Pict.r	Type declaration for PICT resources for debugging PICTs.

Table 8-1 Examples of standard type declaration files

Structure of a Resource Description File

A resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler (the MPW DeRez tool) have no built-in resource types. You need to define your own types or include the appropriate `.r` files.

Table 8-2 shows the statements that a resource description file may contain. Each of the statements is described in the sections following the table.

Statement	Description
`data`	Specify raw data
`type`	Type declaration — declare resource type descriptions for subsequent resource statements
`resource`	Data specification — specify data for a resource type declared in a previous type statement

Table 8-2 Statements in a resource description file

A type declaration describes how the declaration of a resource of that type will look. You must declare a type (with a `type` statement) before you make a resource of that type (with a `resource` statement). Otherwise, you can freely mix `type` and `resource` statements in a file. You can redefine a type any number of times, even a type defined in the standard type declaration files described in Table 8-1, as long as the Redeclared types are OK option is set on the Symantec Rez page of the **Project Options** dialog. (See "Setting Symantec Rez Options" later in this chapter for more information on setting options.)

A resource description file can also include comments and preprocessor directives. Comments can be included any place white space is allowed in a resource description file, by putting them within the comment delimiters `/*` and `*/`. Note that comments do not nest. For example, this is *one* comment:

```
/* Hello /* there */
```

Symantec Rez also supports C++ style comments:

```
type 'tost' {  // Ignore rest of this line.
```

Preprocessor directives substitute macro definitions and include files, and provide if-then-else processing before other Symantec Rez processing takes place. The syntax of the preprocessor is very similar to that of the C language preprocessor.

Sample resource description file

An easy way to learn about the resource description format is to decompile some existing resources. For example, using DeRez to decompile one of an application's resources might generate this:

```
resource 'WIND' (128, "Sample Window") {
   {64, 60, 314, 460},
   documentProc,
   visible,
   noGoAway,
   0x0,
   "Sample Window"
};
```

This resource data corresponds to the following type declaration, contained in `Types.r`:

```
type 'WIND'{
   rect;                        /* bounds     */
   integer documentProc,        /* procID     */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible    */
   fill byte;
   byte noGoAway, goAway;  /* close box */
   fill byte;
   unsigned hex longint;   /* refCon     */
   pstring Untitled="Untitled";
                           /* title      */
   /* . . . */
}
```

Type and resource statements are explained in detail in the reference section that follows.

Resource Description Statements

This section describes the syntax and use of the three types of resource description statements available for the resource compiler: `data`, `type`, and `resource`.

Syntax notation

Table 8-3 shows the syntax notation used with resource description statements.

Notation	Description
`terminal`	Plain text indicates a word that must appear in the statement exactly as shown. Special symbols (such as, `-`, `*`, `=`) and punctuation (such as "`,`" and "`;`") must also be entered exactly as shown.
nonterminal	Items in italics can be replaced by anything that matches their definition.
[*optional*]	Square brackets mean that the enclosed elements are optional.
repeated ...	An ellipsis (...), when it appears in the text of this reference only, indicates that the preceding item can be repeated one or more times.
a | *b*	A vertical bar indicates an either/or choice.
(*grouping*)	Parentheses indicate grouping (useful with the | and ... notation).
'[' *x* ']'	Curly single quotation marks ('...') indicate that one of the syntax notation characters (for example, [or]) must be written as a literal. In this example, the brackets would be typed literally. They do *not* mean that the *x* is optional.

Table 8-3 Syntax notation

Spaces between syntax elements, constants, and punctuation are optional. They are used for readability only.

Syntax elements in resource description statements may be separated by spaces, tabs, returns, or comments.

Special terms

The terms in Table 8-4 represent a minimal subset of the symbols used to describe the syntax of commands in the resource description language.

Expression is defined in the section "Expressions" later in this chapter.

Term	Definition
resource-type	long-expression
resource-name	string
resource-ID	word-expression
ID-range	*ID* [: *ID*]

Table 8-4 Special terms

A full syntax definition can be found at the end of this chapter.

Data—specify raw data

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

Syntax

`data` *res-type* '(' *ID* [, *resource-name*] [, *attributes...*] ')' '{'
data-string
'}' ;

Description

A data statement reads the data found in *data-string* and writes it as a resource with the type *res-type* and the ID *ID*. You can optionally specify a resource name, resource attributes, or both.

For example,

```
data 'PICT' (128) {
    $"4F35FF8790000000"
    $"FF234F35FF790000"
};
```

Type—declare resource type

A type declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one type declaration is given for a resource type and redeclared types are allowed, the last one read before the data definition is the one that's used. This lets you override declarations from include files or previous resource description files.

Syntax

```
type  resource-type  [ '(' ID-range ')' ]  '{'
   type-specification...
'}' ;
```

You can also declare a resource type that uses another resource's type declaration by using the following variant of the type statement:

```
type  resource-type1  [ '(' ID-range ')' ]
   as  resource-type2  [ '(' ID ')' ] ;
```

Description

A type declaration causes any subsequent `resource` statement for the type *resource-type* to use the declaration {*type-specification...*}. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Table 8-5 lists the data-type specifications for the `type` statement.

Specification	Description
`bitstring[`*n*`]`	Bitstring of n bits
`byte`	8-bit number
`integer`	16-bit number
`longint`	32-bit number
`Boolean`	1-bit Boolean value
`char`	8-bit character
`string`	Plain (no length indicator or termination string character)
`pstring`	Pascal string
`wstring`	Word string
`cstring`	C string
`point`	Point
`rect`	Rect
`fill`	Zero fill
`align`	Zero fill to nibble, byte, word, or long word boundary
`switch`	Control construct (case statement)
`array`	Array data specification—zero or more instances of data types

Table 8-5 Data-type specifications for type statement

These types can be used singly or together in a `type` statement. Each of these type specifiers is described in the sections that follow.

Note

Several of these types require additional fields. The exact syntax is given in the sections that follow.

Data-type specifications

A data-type statement declares a field of the given data type. It can also associate symbolic names or constant values with the data type. The data-type specification can take three forms, as shown in this example:

```
type 'XAMP' {
   /* declare a resource of type 'XAMP' */
   byte;
   byte off=0, on=1;
   byte = 2;
};
```

The first `byte` statement declares a byte field; the actual data is supplied in a subsequent `resource` statement.

The second `byte` statement is identical to the first, except that the two symbolic names `off` and `on` are associated with the values `0` and `1`. These symbolic names could be used in the `resource` data.

The third `byte` statement declares a byte field whose value is always `2`. In this case, no corresponding statement would appear in the `resource` data.

Numeric expressions and strings can appear in `type` statements; they are defined in the section "Expressions" later in this chapter.

Numeric types. The numeric types (`bitstring`, `byte`, `integer`, `longint`) are fully specified like this:

```
[ unsigned ]   [ radix ]   numeric-type
   [ = expr | symbol-definition... ];
```

The `unsigned` prefix signals a resource decompiler, such as MPW DeRez, that the number should be displayed without a sign—that the high-order bit can be used for data and the value of the integer cannot be negative. The `unsigned` prefix is ignored by Symantec Rez. Symantec Rez uses a sign if it is specified in the data. Precede a

signed negative constant with a minus sign (–); `0xFFFFFF85` and `-0x7B` are equivalent in value.

Radix is one of the following string constants:

```
hex             decimal         octal
binary          literal
```

You can supply numeric data as decimal, octal, hexadecimal, binary, or literal data.

Table 8-6 lists the numeric type specifiers.

Type	Description
`bitstring '[`*'length'*`]'`	Declare a bitstring of *length* bits (maximum 32)
`byte`	Declare a byte (8-bit) field (this is the same as `bitstring[8]`)
`integer`	Integer (16-bit) field (this is the same as `bitstring[16]`)
`longint`	Long integer (32-bit) field (this is the same as `bitstring[32]`)

Table 8-6 Numeric type specifiers

Symantec Rez uses integer arithmetic and stores numeric values as integer numbers. Symantec Rez translates `boolean`, `byte`, `integer`, and `longint` values to their `bitstring` equivalents. All computations are done in 32-bits and truncated.

An error is generated if a value won't fit in the number of bits defined for the type. Table 8-7 shows the valid ranges for values of `byte`, `integer`, and `longint` constants.

Type	Maximum	Minimum
`byte`	`255`	`-128`
`integer`	`65,535`	`-32,768`
`longint`	`4,294,967,295`	`-2,147,483,648`

Table 8-7 Value ranges for numeric type specifiers

Boolean type. A `Boolean` is a single bit with two possible states: 0 (or `false`) and 1 (or `true`) (`true` and `false` are global predefined identifiers.) `boolean` values are declared as follows:

```
boolean [ = constant | symbolic-value... ];
```

The type `boolean` declares a 1-bit field; this is equivalent to:

```
unsigned bitstring[1]
```

Note

This type is not the same as a `Boolean` variable as defined by Pascal or C/C++.

Character type. Characters are declared as follows:

```
char  [ = string | symbolic-value... ];
```

Type `char` declares an 8-bit field (this is the same as writing `string[1]`).

Here is an example:

```
type 'SYMB'  {
   char  dollar = "$", percent = "%";
};

resource 'SYMB' (128)  {
   dollar
};
```

String type. String data types are specified as:

string-type `['[' ` *length* `']'] [=` *string* `|` *symbol-value...*`];`

Table 8-8 lists the string type specifiers.

String type	Description
`[hex] string`	Plain string that has no length indicator or termination character generated. The optional hex prefix tells a resource decompiler, like MPW DeRez, to display it as a hex string. `string[`*n*`]` contains *n* characters and is *n* bytes long. The type `char` is shorthand for `string[1]`.
`pstring`	Pascal string has a leading byte containing the length information. `pstring[`*n*`]` contains *n* characters and is *n*+1 bytes long. `pstring` has a built-in maximum length of 255 characters, the highest value the length byte can hold. If the string is too long to fit the field, a warning is given and the string is truncated.
`wstring`	Word string is a very large `pstring`; its length is stored in the first two bytes. Therefore, a word string can contain up to 65,535 characters. `wstring[`*n*`]` contains n characters and is *n*+2 bytes long.
`cstring`	C string generates a trailing null byte. `cstring[`*n*`]` contains *n*−1 characters and is *n* bytes long. A C string of length 1 can be assigned only the value `""`, because `cstring[1]` has room only for the terminating null.

Table 8-8 String type specifiers

Each string type may be followed by an optional *length* indicator in brackets (`[`*n*`]`). *Length* is an expression indicating the string length in bytes. *Length* is a positive number in the range `1` ≤ *length* ≤ `2,147,483,647` for `string` and `cstring`, `1` ≤ *length* ≤ `255` for `pstring`, and `1` ≤ *length* ≤ `65,535` for `wstring`.

Note

You cannot assign the value of a literal to a string type.

If no length indicator is given, a `pstring`, `wstring`, or `cstring` stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right with nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

Point and rectangle types. Because points and rectangles appear so frequently in resource files, they have their own simplified syntax:

```
point [ = point-constant | symbolic-value... ];
rect [ = rect-constant | symbolic-value... ];
```

where

point-constant = '{' *x-integer-expr*, *y-integer-expr* '}'

and

rect-constant = '{' *integer-expr*, *integer-expr*,
integer-expr, *integer-expr* '}'

These `type` statements declare a point (two 16-bit signed integers) or a rectangle (four 16-bit signed integers). The integers in a rectangle definition specify the rectangle's upper-left and lower-right points, respectively.

Fill and align types

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The `fill` and `align` type specifiers are two ways of padding fields so that they begin on a boundary that corresponds to the field type. `align` is automatic and `fill` is explicit. Both `fill` and `align` generate zero-filled fields.

Fill specification. The `fill` statement causes Symantec Rez to add the specified number of bits to the data stream. The fill is always `0`. The form of the statement is

```
fill fill-size [ '[' length ']' ] ;
```

where *fill-size* is one of the following strings:

```
bit             nibble          byte
word            long
```

These declare a fill of `1`, `4`, `8`, `16`, or `32` bits (optionally multiplied by the *length* modifier). *Length* is an expression ≤ `2,147,483,647`.

The following fill statements are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

The full form of a `type` statement specifying a fill might be:

```
type 'XRES' {
   data-type specifications;
   fill bit[2];
};
```

Align specification. Alignment causes Symantec Rez to add fill bits of zero value until the data is aligned at the specified boundary. An alignment statement takes the following form:

```
align  align-size ;
```

where *align-size* is one of these strings:

```
nibble          byte            word
long
```

Alignment pads the data with zeros until the data is aligned on a 4-, 8-, 16-, or 32-bit boundary. This alignment affects all data from the point where the alignment is specified until the alignment is changed by another `align` statement.

Array type
An array is declared as follows:

```
[wide] array [ array-name | '[' length ']' ]  '{'
   array-list
'}';
```

The *array-list*, a list of type specifications, is repeated zero or more times. The wide option outputs the array data in a wide display format (in a resource decompiler like MPW DeRez)—the elements that make up the array-list are separated by a comma and space instead of a comma, Return, and Tab. This has no effect in Symantec Rez. Either *array-name* or `[`*length*`]` may be specified. *Array-name* is an identifier.

If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$CountOf(`*array-name*`)`

function; otherwise, a resource decompiler like MPW DeRez will treat the array as open-ended. For example,

```
type 'STR#' {
   /* define a string list resource */
   integer = $$Countof(StringArray);
   array StringArray {
      pstring;
   };
};
```

The `$$CountOf()` function returns the number of array elements (in this case, the number of strings) from the resource data.

If `[`*length*`]` is specified, there must be exactly *length* elements.

Array elements are generated by commas, which are element separators. Semicolons are element terminators. In this example, however, it may be a good idea to use semicolons as element separators:

```
type 'xyzy' {
   array Increment {
      integer = $$ArrayIndex(Increment);
   };
};

resource 'xyzy' (0) {
   {  /* zero elements */
   }
};

resource 'xyzy' (1) {
   {  /* two elements */
      ,
   }
};

resource 'xyzy' (3) {
   }  /* two elements */
      ;;
   }
};

/* The only way to specify one element in
 * an array that has all constant elements,
 * is to use a semicolon terminator.
 */
```

```
resource 'xyzy' (4) {
   {  /* one element */
      ;
   }
};
```

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is:

`switch '{'` *case-statement...* `'}';`

where a *case-statement* has this form:

`case` *case-name* `:` `[` *case-body* `;` `]...`

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

`key` *data-type* `=` *constant*

The case that applies is based on the key value. For example,

```
type 'DITL'{
   /* dialog item list declaration
    * from Types.r
    */

   ...type specifications...

   switch {    /* one of the following */
   case Button:
      boolean enabled, disabled;
      key bitstring[7] = 4; /* key value */
      pstring;
   case CheckBox:
      boolean enabled, disabled;
      key bitstring[7] = 5; /* key value */
      pstring;

   ...and so on.

   };
};
```

Sample type statement

The following sample `type` statement is the standard declaration for a `'WIND'` resource, taken from the `Types.r` file:

```
type 'WIND'{
   rect;                        /* bounds     */
   integer documentProc,        /* procID     */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible    */
   fill byte;
   byte noGoAway, goAway;   /* close box */
   fill byte;
   unsigned hex longint;    /* refCon     */
   pstring Untitled="Untitled";
                            /* title      */
   /* . . . */
}
```

The type declaration consists of header information followed by a series of statements, each terminated by a semicolon (`;`). The header of the sample window declaration is

```
type 'WIND'
```

The header begins with the `type` keyword followed by the name of the resource type being declared—in this case, a window. You may specify a standard Macintosh resource type, as shown in *Inside Macintosh: More Macintosh Toolbox, Chapter 1*, or you may declare a resource type specific to your application.

The left brace (`{`) introduces the body of the declaration. The declaration continues for as many lines as necessary until a matching right brace (`}`) is encountered. You can write more than one statement on a line, and a statement may cover more than one line (like the `integer` statement above). Each statement represents a field in the resource data. Recall that comments may appear wherever white space appears in the resource description file; comments begin with `/*` and end with `*/`, as in C.

Symbol definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

```
name = value   [, name = value   ]...
```

For numeric data, the "= *value*" part of the statement can be omitted. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out—if *value* is omitted, it's assumed to be one greater than the previous value. (The value is assumed to be zero if it's the first value in the list.) This is true for `bitstrings` and their derivatives: `byte`, `integer`, and `longint`. For example,

```
integer  documentProc, dBoxProc, plainDBox,
         altDBoxProc, noGrowDocProc,
         zoomProc=8, rDocProc=16;
```

In this example, the symbolic names `documentProc`, `dBoxProc`, `plainDBox`, `altDBoxProc`, and `noGrowDocProc` are automatically assigned the numeric values `0`, `1`, `2`, `3`, and `4`.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value. For example:

```
integer  documentProc=0, dBoxProc=1,
         plainDBox=2, altDBoxProc=3,
         rDocProc=16, Document=0, Dialog=1,
         DialogNoShadow=2,
         ModelessDialog=3,
         DeskAccessory=16;
```

Resource—specify resource data

`Resource` statements specify actual resources, based on previous type declarations.

Syntax

`resource` *res-type* '(' *ID* [, *resource-name*] [, *attributes*] ')' '{'
[*data-statement* [, *data-statement*]...]
'}';

Description

A `resource` statement specifies the data for a resource of type *res-type* and ID *ID*. The latest type declaration declared for *res-type* is used to parse the data specification. *Data-statements* specify the actual data; *data-statements* appropriate to each resource type are defined in the next section.

The resource definition causes an actual resource to be generated. A `resource` statement can appear anywhere in the resource description file, or even in a separate file specified on the command

line or as an #include file, as long as it comes after the relevant type declaration.

Data statements
The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration. Table 8-9 lists the data statement specifications.

Base type	Instance types
string	string, cstring, pstring, wstring, char
bitstring	boolean, byte, integer, longint, bitstring
rect	rect
point	point

Table 8-9 Data statement specifications

Switch data. Switch data statements are specified by using this format:

```
switch-name   data-body
```

For example, the following could be specified for the 'DITL' type given earlier:

```
...
CheckBox { enabled, "Check here" },
...
```

Array data. Array data statements have this format:

```
'{' [ array-element [ , array-element ]... ] '}'
```

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the 'STR#' resource defined earlier:

```
resource 'STR#' (280) {
    {  "this",
       "is",
       "a",
       "test"
    }
};
```

Sample resource definition

This section describes a sample resource description file for a window. Here, again, the type declaration is given in the section "Sample type statement" earlier in this chapter.

See Inside Macintosh: Macintosh Toolbox Essentials, *"Window Manager," for information about resources in windows.*

```
type 'WIND'{
   rect;                        /* bounds     */
   integer documentProc,        /* procID     */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible    */
   fill byte;
   byte noGoAway, goAway;  /* close box */
   fill byte;
   unsigned hex longint;   /* refCon     */
   pstring Untitled="Untitled";
                           /* title      */
   /* . . . */
}
```

Here is a typical example of the window data corresponding to this declaration:

```
resource 'WIND' (128, "My window",
                 preload) {
                   /* Status window      */
   {40,80,120,300},/* Bounding rectangle */
   documentProc,   /* documentProc etc.. */
   Visible,        /* Visible or         */
                   /*   Invisible        */
   goAway,         /* GoAway or NoGoAway */
   0,              /* Reference value    */
                   /*   RefCon           */
   "Status Report" /* Title              */
   };
```

This data definition declares a resource of type `'WIND'`, using whatever type declaration was previously specified for `'WIND'`. The resource ID is `128`; the resource name is `"My window"`. Because the resource name is represented by the Resource Manager as a `pstring`, it should not contain more than 255 characters. The resource name may contain any character, including the null character (`'\000'`). The resource will be loaded when the resource file is opened.

The first statement in the window type declaration declares a bounding rectangle for the window:

```
rect;
```

The rectangle is described by two points: the upper-left corner and the lower-right corner. The points of a rectangle are separated by commas like this:

```
{ top, left, bottom, right }
```

An example of data for these coordinates is

```
{ 40, 80, 120, 300 }
```

Symbolic names. Symbolic names may be associated with particular values of a numeric type. Notice that a symbolic name is given for the data in the second, third, and fourth fields of the window declaration. For example,

```
integer  documentProc=0, dBoxProc=1,
         plainDBox=2, altDBoxProc=3,
         noGrowDocProc=4, zoomProc=8,
         rDocProc=16;       /* windowType */
```

This statement specifies a signed 16-bit integer field with symbolic names associated with the values 0 to 4, 8, and 16. The values 0 through 4 need not be indicated in this case; if no values are given, symbolic names are automatically given values starting at 0, as explained previously.

In the sample window declaration, the values `True` (`1`) and `False` (`0`) were given to two different byte variables. For clarity, those symbolic names were used in the window's resource data. That is:

```
visible,
goAway,
```

were used instead of their equivalents:

```
TRUE,
TRUE,
```

or

```
1,
1,
```

Labels

Labels support some of the more complicated resources, such as `'NFNT'` and color QuickDraw resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels.

Syntax

label ::= *character* {*alphanum*}... ':'
character ::= '_' | A | B | C ...
number ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum ::= *character* | *number*

Description

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear in a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See the section "Declaring labels within arrays" on the next page for more information.

The value of a label is always the offset, *in bits*, from the beginning of the resource and the position at which the label occurs when mapped to the resource data. In this example,

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};

resource 'cool' (8) {
  "Neato"
}
```

the integer following the `cstring` would contain:

```
( len("Neato") [5] + null byte [1] ) *
8 [bits per byte]
   = 48.
```

Built-in functions to access resource data

In some cases, accessing the actual resource data to which a label points is desirable. Several built-in functions allow access to that data:

`$$BitField(`*label, startingPosition, numberOfBits*`)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(`*label*`)`
Returns the byte found at *label*.

`$$Word(`*label*`)`
Returns the word found at *label*.

`$$Long(`*label*`)`
Returns the longword found at *label*.

For example, the resource type `'STR '` could be redefined without using a `pstring`. Here is the definition of `'STR '` from `Types.r`:

```
type 'STR ' {
   pstring;
}
```

Here is a redefinition of `'STR '` using labels:

```
type 'STR ' {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop:;
};
```

Declaring labels within arrays

Labels declared within arrays may have many values. For every element in the array, there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to *n* where *n* is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 'test' {
   integer = $$CountOf(array1);
   array array1 {
      integer = $$CountOf(array2);
      array array2 {
foo:     integer;
      };
   };
};
```

```
resource 'test' (128) {
   {
      {1,2,3},
      {4,5}
   }
};
```

In the above example, the label `foo` takes on values listed below:

```
foo[1,1] = 32      $$Word(foo[1,1]) = 1
foo[1,2] = 48      $$Word(foo[1,2]) = 2
foo[1,3] = 64      $$Word(foo[1,3]) = 3
foo[2,1] = 96      $$Word(foo[2,1]) = 4
foo[2,2] = 112     $$Word(foo[2,2]) = 5
```

This built-in function may be helpful in using labels within arrays:

`$$ArrayIndex(`*arrayname*`)`
This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label limitations

Keep in mind that Symantec Rez is basically a one pass compiler. This characteristic explains some of the limitations of labels.

Note

To decompile a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can contain only one undefined label:

```
type 'test' {
    /* In the expression below,
     * start is defined,
     * next is undefined.
     */
start:    integer = next - start;
    /* In the expression below,
     * next is defined because it was
     * used in a previous expression,
     * but final is undefined.
     */
middle:   integer = final - next;
next:     integer;
final:
};
```

Actually, Symantec Rez can compile types that have expressions containing more than one undefined label, but a resource decompiler like DeRez cannot decompile those resources. It simply generates data resource statements.

Note

The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

Using labels: two examples

The first example shows the `'ppat'` declaration using labels. Without using labels, the whole end section of the resource would have to be combined into a single hex string (everything following the `PixelData` label). Using labels, the complete `'ppat'` definition can be expressed in the Rez language.

```
/* PixPat record */

type 'ppat' {
    integer oldPattern, newPattern, ditherPattern;
                        /* Pattern type           */
    unsigned longint = PixMap / 8;
                        /* Offset to pixmap       */
    unsigned longint = PixelData / 8;
                        /* Offset to data         */
    fill long;          /* Expanded pixel image   */
```

```
   fill word;          /* Pattern valid flag      */
   fill long;          /* expanded pattern        */
   hex string [8];     /* old-style pattern       */


/* PixMap record */
PixMap:
   fill long;          /* Base address            */
   unsigned bitstring[1] = 1;
                       /* New pixMap flag         */
   unsigned bitstring[2] = 0;
                       /* Must be 0               */
   unsigned bitstring[13];
                       /* Offset to next row      */
   rect;               /* Bitmap bounds           */
   integer;            /* pixMap vers number      */
   integer unpacked;   /* Packing format          */
   unsigned longint;   /* size of pixel data      */
   unsigned hex longint;
                       /* horizontal resolution   */
                       /* (ppi) (fixed)           */
   unsigned hex longint;
                       /* vertical resolution     */
                       /* (ppi) (fixed)           */
   integer chunky, chunkyPlanar, planar;
                       /* Pixel storage format    */
   integer;            /* # bits in pixel         */
   integer;            /* # bits per field        */
   integer;            /* # components in pixel   */
   unsigned longint;   /* Offset to next plane    */
   unsigned longint = ColorTable / 8;
                       /* Offset to color table   */
   fill long;          /* Reserved                */

PixelData:
   hex string [(ColorTable - PixelData) / 8];

   ColorTable:
      unsigned hex longint;
                          /* ctSeed               */
      integer;            /* transIndex           */
      integer = $$Countof(ColorSpec) - 1;
                          /* ctSize               */
      wide array ColorSpec {
         integer;              /* value           */
         unsigned integer;     /* RGB: red        */
         unsigned integer;     /* green           */
         unsigned integer;     /* blue            */
      };
   };
```

Here is another example of a resource definition that uses labels. In this example, the `$$BitField()` function is used to access information stored in the resource, in order to calculate the size of the various data areas added at the end of the resource. Without labels, all data would have to be combined into one hex string.

```
/* IconPMap (pixMap) record */

type 'cicn' {
   fill long;          /* Base address              */
   unsigned bitstring[1] = 1;
                       /* New pixMap flag           */
   unsigned bitstring[2] = 0;
                       /* Must be 0                 */
pMapRowBytes: unsigned bitstring[13];
                       /* Offset to next row        */
Bounds: rect;          /* Bitmap bounds             */
   integer;            /* pixMap vers number        */
   integer unpacked;   /* Packing format            */
   unsigned longint;   /* Size of pixel data        */
   unsigned hex longint;
                       /* horizontal resolution     */
                       /* (ppi) (fixed)             */
   unsigned hex longint;
                       /* vertical resolution       */
                       /* (ppi) (fixed)             */
   integer chunky, chunkyPlanar, planar;
                       /* Pixel storage format      */
   integer;            /* # bits in pixel           */
   integer;            /* # components in pixel     */
   integer;            /* # bits per field          */
   unsigned longint;   /* Offset to next plane      */
   unsigned longint;   /* Offset to color table     */
   fill long;          /* Reserved                  */

   /* IconMask (bitMap) record */
   fill long;          /* Base address              */
maskRowBytes: integer;
                       /* Row bytes                 */
   rect;               /* Bitmapds                  */

   /* IconBMap (bitMap) record */
   fill long;          /* Base address              */
iconBMapRowBytes: integer;
                       /* Row bytes                 */
   rect;               /* Bitmap bounds             */
   fill long;          /* Handle placeholder        */
```

```
    /* Mask data */
    hex string [$$Word(maskRowBytes) *
       ( $$BitField(Bounds, 32, 16)  /* bottom */
         - $$BitField(Bounds, 0, 16) /* top    */
       )];

    /* BitMap data */
    hex string [$$Word(iconBMapRowBytes) *
       ($$BitField(Bounds, 32, 16)   /* bottom */
         - $$BitField(Bounds, 0, 16) /* top    */
       )];

    /* Color Table */
    unsigned hex longint;    /* ctSeed           */
    integer;                 /* transIndex       */
    integer = $$Countof(ColorSpec) - 1;
                             /* ctSize           */
    wide array ColorSpec {
       integer;              /* value            */
       unsigned integer;     /* RGB: red         */
       unsigned integer;     /* green            */
       unsigned integer;     /* blue             */
    };

    /* PixelMap data */
    hex string [$$BitField(pMapRowBytes,0,13) *
        ($$BitField(Bounds,32,16)    /* bottom */
          - $$BitField(Bounds, 0, 16)/* top    */
        )];
};
```

Preprocessor Directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other Symantec Rez processing takes place.

The syntax of the preprocessor is very similar to that of the C language preprocessor. Preprocessor directives must observe these rules and restrictions:

1. Each preprocessor statement must be expressed on a single line, beginning on a new line and terminated by a return character.

2. The pound sign (#) must be the first character on the line of the preprocessor statement (except for spaces and tabs).

3. Identifiers (used in macro names) may be letters (`A-Z`, `a-z`), digits (`0-9`), or the underscore character (_).

4. Identifiers may be any length.

5. Identifiers may not start with a digit.

6. Identifiers are not case sensitive.

Variable definitions

The `#define` and `#undef` directives let you assign values to identifiers:

```
#define   macro   data
#undef    macro
```

The `#define` directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the Symantec Rez escape character. For example,

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

(Quotation marks within strings must also be escaped.)

`#undef` removes the previously defined identifier *macro*. You can also define and undefine macros in the Prefix String section of the Symantec Rez page of the **Project Options** dialog, described later in this chapter. Table 8-10 lists the macros that are predefined in Symantec Rez.

Variable	Value	Description
`true`	`1`	Use for the Boolean value true
`false`	`0`	Use for the Boolean value false
`rez`	`1`	Use to test whether a resource compiler is running
`derez`	`0`	Use to test whether a resource de-compiler is running
`Symantec_Rez` `THINK_Rez`	`1`	Use to test which resource compiler is running

Table 8-10 Predefined macros

Header file processing

You can include the text of another file in your file with the `#include` directive. It works like the `#include` directive in C:

```
#include file
```

Include the text file *file*. The maximum nesting is to ten levels. For example,

```
#include "MyTypes.r"
```

You can enclose the file name in either quotes (`" "`) or angle brackets (<>). Table 8-11 shows the rules Symantec Rez uses to find header files.

#include statement	Symantec Rez...
`<filename.h>`	Looks only in the system tree
`"filename.h"`	Looks first in the referencing folder, then in the project tree, and finally in the system tree.

Table 8-11 Header file search rules

The referencing folder is the one that contains the file that has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h` and that file in turn has the line `#include "MyUtilTypes.h"`, Symantec Rez will look for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

Apple's MPW Rez does not support angle brackets in the `#include` directive. Instead, you enclose all file names in quotes. If you plan to use your code with both Symantec Rez and Apple's MPW Rez, test for which compiler is running with the `Symantec_Rez` macro, like this:

```
#ifdef Symantec_Rez
#include <filename.h>
#else
#include "filename.h"
#endif
```

Pragma once directives

You may want to create a header file that you want included in several places but which should define its symbols only once in a file. You can use the `#pragma once` directive to do this.

If you have the directive:

```
#pragma once
```

in your header file, Symantec Rez will include that file only once. If another file tries to include that header file, Symantec Rez knows that the symbols in that file have already been defined, so it doesn't process the file again.

Keep these restrictions in mind when you use `#pragma once`:

- It's case sensitive. For example, if you include a file once with the statement `#include <myres.h>` and later with the statement `#include <MyRes.h>`, Symantec Rez will include it twice.
- It doesn't distinguish between files included with `<...>` and `"..."`. For example, suppose you have two header files named `xyz.h`; one is in the system tree and the other is in the project tree. If you include one with `#include <xyz.h>` and another with `#include "xyz.h"`, Symantec Rez will not include the second file.
- It ignores characters after the first 32. If you include two files with names that start with the same 32 characters, Symantec Rez will not include the second file.

If-then-else processing

These directives provide conditional processing.

```
#if expression
[  #elif expression ]
[  #else ]
#endif
```

Expression is defined in the section "Expressions" later in this chapter. When used with the `#if` and `#elif` directives, *expression* may also include this expression:

```
defined identifier
```

or

```
defined '(' identifier ')'
```

The following may also be used in place of `#if`:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource  'STR '  (199)  {
#ifdef English
   "Hello"
#elif defined (French)
   "Bonjour"
#elif defined (Thai)
   "Sawati"
#elif defined (Japanese)
   "Konnichiwa"
#endif
};
```

Resource Description Syntax

This section describes the details of the resource description syntax.

Numbers and literals

All arithmetic is performed as 32-bit signed arithmetic. The basic constants are:

- Decimal (*nnn...*): Signed decimal constant between 4,294,967,295 and -2,147,483,648.
- Hex (`0X`*hhh...* or `$`*hhh...*): Signed hexadecimal constant between `0X7FFFFFFF` and `0X80000000`.
- Octal (`0`*ooo...*): Signed octal constant between `017777777777` and `020000000000`.
- Binary (`0B`*bbb...*): Signed binary constant between `0B11111111111111111111111111111111` and `0B10000000000000000000000000000000`.
- Literal (`'`*aaaa*`'`): A literal may contain one to four characters. Characters are printable ASCII characters or escape characters. If there are less than four characters in the literal, then the characters to the left (high bits) are assumed to be null characters (`'\000'`). Characters that are not in the printable character set (and are not the

characters \' and \\, which have special meanings) can be escaped according to the character escape rules. (See the section "Strings" later in this chapter.)

Literals and numbers are treated in the same way by the resource compiler. A literal is a value within single quotation marks. For instance, 'A' is a number with the value 65; on the other hand, "A" is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that "A" is not a valid number and 'A' is not a valid string.) The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

Literals are padded with nulls on the left side so that the literal 'ABC' is stored as shown in Figure 8-2.

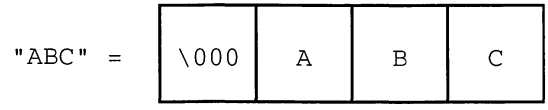

Figure 8-2 Padding of literals

Expressions

An expression may consist of simply a number or literal. Expressions may also include numeric variables, labels, and system functions.

Table 8-12 lists the operators in order of precedence with highest precedence first. Groupings indicate equal precedence. Evaluation is always left to right when the priority is the same. Variables are defined following the table.

Operator	Meaning
(*expr*)	Use to force precedence in expressions
-expr	Arithmetic (two's complement) negation of *expr*
~expr	Bitwise (one's complement) negation of *expr*
!expr	Logical negation of *expr*
expr1 * *expr2*	Multiplication

Table 8-12 Operator precedence

Operator	Meaning
expr1 / *expr2*	Division
expr1 % *expr2*	Remainder from dividing *expr1* by *expr2*
expr1 + *expr2*	Addition
expr1 - *expr2*	Subtraction
expr1 << *expr2*	Shift left—shift *expr1* left by *expr2* bits
expr1 >> *expr2*	Shift right—shift *expr1* right by *expr2* bits
expr1 > *expr2*	Greater than
expr1 >= *expr2*	Greater than or equal to
expr1 < *expr2*	Less than
expr1 <= *expr2*	Less than or equal to
expr1 == *expr2*	Equal
expr1 != *expr2*	Not equal
expr1 & *expr2*	Bitwise AND
expr1 ^ *expr2*	Bitwise XOR
expr1 | *expr2*	Bitwise OR
expr1 && *expr2*	Logical AND
expr1 || *expr2*	Logical OR

Table 8-12 Operator precedence *(Continued)*

The logical operators `!`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&`, and `||` evaluate to `1` (true) or `0` (false).

Variables and functions

Some resource compiler variables contain commonly used values. All Symantec Rez variables start with `$$` followed by an alphanumeric identifier.

The following variables and functions have string values (typical values are given in parentheses):

`$$Date`
Current date. Useful for putting timestamps into the resource file. The format is generated through the ROM call `IUDateString()`. (`"Thursday, May 20, 1987"`)

`$$Format("`*formatString*`", `*arguments*`)`
Works just like the `#printf` directive except that `$$format()` returns a string rather than printing to standard output.

`$$Name`
Name of resource from the current resource. The current resource is the resource being generated in a `resource` statement, or being included from an `include` statement.

For example, to include all `'DRVR'` resources from one file and keep the same information, but also set the `SYSHEAP` attribute:

```
INCLUDE "file" 'DRVR' (0:40) AS
 'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

The `$$Type`, `$$ID`, `$$Name`, and `$$Attributes` variables are undefined outside of a `change`, `delete`, `include`, or `resource` statement.

`$$Time`
Current time. Useful for time-stamping the resource file. The format is generated through the ROM call `IUTimeString()`.
`("7:50:54 AM")`

`$$Version`
Version number of Symantec Rez. `("V3.0")`

These variables and functions have numeric values:

`$$Attributes`
Attributes of resource from the current resource. See the `$$Name` string variable.

`$$BitField(`*label*`, `*startingPosition*`, `*numberOfBits*`)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(`*label*`)`
Returns the byte found at *label*.

`$$Day`
Current day. Range 1–31.

`$$Hour`
Current hour. Range 0–23.

`$$ID`
ID of resource from the current resource. See the `$$Name` string variable.

`$$Long(`*label*`)`
Returns the longword found at *label*.

`$$Minute`
Current minute. Range 0–59.

`$$Month`
Current month. Range 1–12.

`$$PackedSize(`*Start*, *RowBytes*, *RowCount*`)`
Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the Toolbox routine `UnpackBits()` *RowCount* times. `$$PackedSize()` returns the unpacked size of the data found at *start*. Use this function only for decompiling resource files. An example of this function is found in `Pict.r`.

`$$ResourceSize`
Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource. (See the `'KCHR'` resource in `SysTypes.r` for an example.)

`$$Second`
Current second. Range 0–59.

`$$Type`
Type of resource from the current resource. See the `$$Name` string variable.

`$$Weekday`
Current day of the week. Range 1–7 (that is, Sunday–Saturday).

`$$Word(`*label*`)`
Returns the word found at *label*.

`$$Year`
Current year.

Strings

There are two basic types of strings:

- Text string ("*a...*"): The string can contain any printable character except ' " ' and '\'. These and other characters can be created through escape sequences. (See the section "Escape characters" on page 185.) The string "" is a valid string of length 0.
- Hex string ($"*bb...*"): Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string $"" is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

Figure 8-3 shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello".
```

\005	H	e	l	l	o	\000	\000	\000	\000	\000

Figure 8-3 Internal representation of a Pascal string

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma or brace) signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks ("") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as:

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash like this:

```
"Hello \"out\" there."
```

Escape characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence \n.

Table 8-13 shows the valid escape sequences.

Name	Escape Sequence	Hex Value
Tab	\t	$09
Backspace	\b	$08
Return	\r	$0A
Newline	\n	$0D
Form feed	\f	$0C
Vertical tab	\v	$0B
Rubout	\?	$7F
Backslash	\\	$5C
Single quotation mark	\'	$3A
Double quotation mark	\"	$22

Table 8-13 Escape sequences

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are given in Table 8-14.

Base	Form (# digits)	Example
2	\0B*bbbbbbbb* (8)	\0B01000001
8	*ooo* (3)	\101
10	\0D*ddd* (3)	\0D065
16	\0X*hh* (2)	\0X41
16	\$*hh* (2)	\$41

Table 8-14 Numeric escape sequences

Here are some examples:

```
\077          /* 3 octal digits          */
\0xFF         /* '0x' + 2 hex digits     */
\$F1\$F2\$F3  /* '$' + 2 hex digits      */
\0d099        /* '0d' + 3 decimal digits */
```

Note

An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write `AB\007CD`, not `AB\7CD`.

Setting Symantec Rez Options

To set options for Symantec Rez, choose **Symantec Rez** from the scrolling list that appears when you choose **Options** from the **Project** menu.

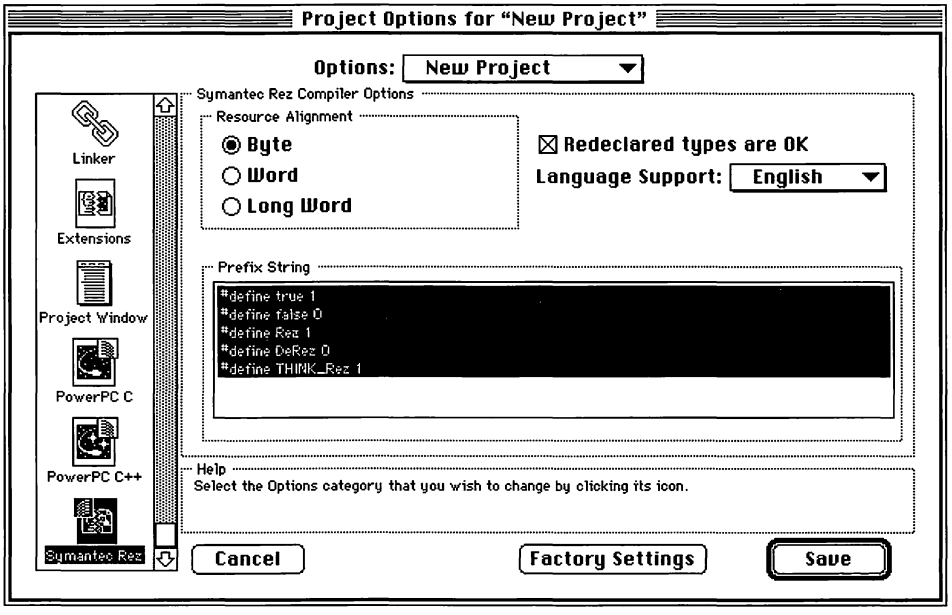

Figure 8-4 The Symantec Rez options dialog

Resource alignment

This option lets you choose how your resources are aligned: along byte, word, or long word boundaries. The default is byte.

Redeclared types are ok

This option lets you choose if Symantec Rez continues compiling your file if you declare a resource type more than once in a resource file. If it's selected, Symantec Rez prints a warning and continues compiling. Otherwise, Symantec Rez prints an error message and stops compiling. The option is selected by default.

Prefix String

The Symantec Rez Prefix String lets you include some text in all the Symantec Rez source files in your project. The effect is the same as if you put the string at the beginning of all your resource description files.

Note

Previous versions of THINK Rez defined the macros `true`, `false`, `Rez`, `DeRez`, and `THINK_Rez` in the prefix. These are defined internally in Symantec Rez.

If you need to define a macro in all your files, define it here. For example, you may have some resources you use for debugging that are compiled only if the macro `DEBUG` is defined. To include that code, include the following line:

```
#define DEBUG
```

When you no longer need to include the debugging code, just delete that line from this page. You don't need to edit every `.r` file in your project.

Note

When you change the Symantec Rez prefix string, the Symantec Project Manager recompiles all the Symantec Rez files in your project.

Language Support

Use this option to accept foreign language double-byte characters in string and character literals, as well as comments. This option causes the compiler to produce localized error messages. Currently, any

supported language may be used for input; however, Japanese is the only supported foreign language for error messages.

Differences from MPW Rez

Symantec Rez is based on Rez, Apple's resource utility that's available with Apple's Macintosh Programmer's Workshop (MPW). Symantec Rez supports most of the Rez features described in the MPW 3.2 documentation.

The most significant difference between MPW Rez and Symantec Rez is that some of MPW Rez's features are handled by the Symantec Project Manager and not by Symantec Rez. Symantec Rez only compiles resource description files; MPW Rez also modifies resource files and copies them into your application. The Symantec Project Manager handles these tasks. To copy a resource file into your application, simply include it in your project and the Symantec Project Manager copies its contents into your application automatically. To modify a resource file, double-click its name in your project window and the Symantec Project Manager launches your resource editor (such as ResEdit or Resourcerer) and opens the file.

The following are *not* supported in Symantec Rez:

- The `change` and `delete` statements
- The `$$shell` function

The following is a feature in Symantec Rez that is not in MPW Rez.

> In Symantec Rez, the `#include` statement uses the same rules for finding header files that Symantec C and Symantec C++ use. For more information, see the section "Header file processing" earlier in this chapter.

Language Reference A

Symantec C is a conforming implementation of the C programming language as defined in the ANSI C standard, *ANSI/ISO 9899— 1990, Information Technology-Programming Language C.* Symantec C++ implements the C++ language as defined in *The Annotated C++ Reference Manual,* currently under review by working group X3J16 of the ANSI standards committee.

This appendix describes how:

- Symantec C implements those aspects of the C language that the ANSI C standard denotes as implementation-defined
- Symantec C++ implements those aspects of the language denoted as implementation-defined in *The Annotated C++ Reference Manual.*

This appendix is divided into two parts: Symantec C Language Reference and Symantec C++ Language Reference. In the C Language Reference, section numbers correspond to the sections in the ANSI standard. In the C++ Language Reference, section numbers correspond to the sections in *The Annotated C++ Reference Manual* (ARM).

Contents

Part I - Symantec C Language Reference

Introduction

This section describes how Symantec C implements the behavior that the ANSI Standard identifies as implementation-defined. It also documents consistent behavior in Symantec C that the standard identifies as "undefined behavior." These behaviors represent conforming extensions.

Note

This section is not a substitute for the ANSI standard.

Implementation-defined behavior

The ANSI Standard defines implementation-defined behavior as:

> "...behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document."

Implementation-defined behavior covers such things as the way error messages are reported, the number of significant characters in identifiers, the format for integers and floating-point numbers, and so on.

Undefined behavior

The ANSI Standard defines undefined behavior as:

> "...behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which the standard imposes no requirements."

In most cases, undefined behavior is ignored, generates a diagnosed error, or results in a run-time error. This section uses the notation "(Conforming Extension)" to mark the instances in which Symantec C behaves in a predictable manner for cases that the standard specifies as undefined. If you are writing portable C code, your program should not rely on behaviors described in Symantec C as conforming extensions.

Setting ANSI conformance

The default options settings for Symantec C are not ANSI-conformant. To make Symantec C an ANSI-conformant compiler, open the **PowerPC C options** dialog, choose the Language Settings

page, and click the ANSI Settings button. Any options that affect ANSI conformance appear on that page.

About the standard libraries

For more information about implementation-defined behavior or Symantec C extensions to the standard libraries described in section 4.0 of the standard, refer to the online *Standard Libraries Reference* that came with your package.

Note

The descriptions of standard libraries, as presented in this appendix, refer to the Symantec libraries, not the Apple libraries.

The section numbers in C Language Reference correspond to the sections in the ANSI standard.

C Language Reference

2.1.1.3 Diagnostics

Errors that occur during translation are reported in the Compile Errors window. The offending line is highlighted in a text editing window. Link errors are reported in a Link Errors window.

2.1.2.2.1 Program startup

If the `ccommand` function is not used, `argc` is set to 1, and `argv[0]` is the empty string. If the `ccommand` function is used, `argc` and `argv` are set according to the values provided in the `ccommand` dialog box. See the description of `ccommand` in the online *Standard Libraries Reference* for more information.

2.1.2.3 Program execution

In programs that use the console package, the Console window is an interactive device. In Macintosh programs (programs that initialize the Macintosh Toolbox), the interactive device is output only.

2.2.1 Character sets

The source and execution characters include the full Macintosh character set.

2.2.1.2 Multibyte characters

There are no shift states for multibyte characters. Multibyte characters are 1 byte long.

2.2.4.2.1 Sizes of integral types <limits.h>

The number of bits in a character in the execution character set is 8.

3.1.2 Identifiers

All characters in an identifier are significant. The number of significant characters in an identifier with external linkage is 1024. In addition, case is significant in identifiers with external linkage.

3.1.2.2 Linkages of identifiers

Identifiers without initializers may be declared later with internal linkage. Any other combination of internal and external linkage is a compile-time diagnostic.

```
extern int foo;       /* this foo has      */
                      /* external linkage */
static int foo = 63; /* this foo has      */
                      /* internal linkage */
```

3.1.2.5 Types

Integers are represented as two's complement binary numbers. The sizes of integer types are:

Type	Bytes
`char`	1
`short`	2
`long`	4
`int`	4

The limits for the integer types are given in the header file `<limits.h>`.

Symantec C uses two different representations for floating-point values. The floating-point representations are:

- 4-byte IEEE single precision
- 8-byte IEEE double precision

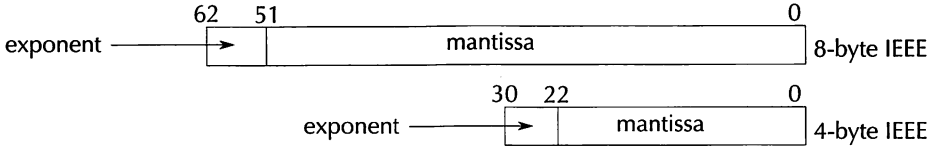

Figure A-1 Floating-point formats

These formats are documented in detail in *Inside Macintosh PowerPC Numerics Manual* (Addison-Wesley) by Apple Computer.

Table A-1 shows how floating-point types map to floating-point formats.

Type	Format
`float`	4-byte IEEE
`long double`	8-byte IEEE
`double`	8-byte IEEE

Table A-1 The floating-point format used for each type

3.1.3.4 Character constants

If a string literal begins with the sequence `\p` or `\P`, it is treated as a Pascal string. The `\p` or `\P` is replaced with the length of the string. The type of Pascal strings is `unsigned char []`, or optionally, `char []`. Pascal strings are not null-terminated.

Characters in the source character set are mapped one-to-one to the execution character set. Multibyte characters are 1 byte long and map one-to-one with single-byte characters.

The basic execution character set consists of all 256 Macintosh characters. There are no integer character constants or escape sequences that cannot be represented in the basic execution character set.

An integer constant may contain one, two, or four characters from the execution character set. All multi-character constants are of type `int`.

The backslash character (`\`) is ignored for all unspecified escape sequences. (Conforming Extension)

3.1.7 Header names

In header name preprocessing, header names are treated as character strings with different delimiters. The characters `'`, `\`, `"`, and `/*` are allowed between the < and > delimiters, and the characters `'`, `\`, and `/*` are allowed between the `"` delimiters. (Conforming Extension)

3.2.1.2 Signed and unsigned integers

If an integer is converted to a shorter signed integer, the low-order bits are retained and the high-order bit of the shorter integer is treated as the sign bit.

If an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented in the signed integer, there is no change in representation, but the high bit becomes the sign bit.

3.2.1.3 Floating and integral

When a value of an integral type is converted to a floating type, if the value being converted is in the range of values that cannot be represented exactly, the result is rounded in the current rounding mode. The default rounding mode is round-to-nearest, and may be changed by the program.

3.2.1.4 Floating types

If a floating value being converted is in the range of values that can be represented, but cannot be represented exactly, the result is rounded in the current rounding mode. The default rounding mode is round-to-nearest, and may be changed by the program.

3.3 Expressions

Bitwise operations on signed integers are carried out as if they were unsigned.

3.3.2.3 Structure and union members

If a member of a union object is accessed using a member of a different type, the data stored at that location is treated as if it were a member of the accessing type.

3.3.3.4 The sizeof operator

The type of the `sizeof` operator is `size_t`, which is defined as `unsigned int`.

3.3.4 Cast operators

A pointer to a function may be converted to a pointer to an object without losing the value of the pointer. (Conforming Extension)

When converting from a pointer to an integer, if the integer is 4 bytes, then all the bits are used without a change of representation; the 4-byte quantity is interpreted as an integer whose type (for

example, signed or unsigned) is given by the cast expression. If the integer is smaller than a pointer, then the low-order bytes are used.

3.3.5 Multiplicative operators

The sign of the remainder is the same as the sign of the dividend. Table A-2 shows the results of the division and modulus operators.

If a and b are...		Then a/b and a%b are...	
a=	b=	a/b=	a%b=
5	3	1	2
-5	3	-1	-2
5	-3	-1	2
-5	-3	1	-2

Table A-2 Results of division and modulus operators

3.3.6 Additive operators

Adding to or subtracting from a pointer that does not behave like a pointer to an element of an array is allowed. Memory is treated as a linear address space. Pointers that do not behave as if they point to the same array object may be subtracted. (Conforming Extension)

The type of integer required to hold the difference between two pointers is `ptrdiff_t`, which is defined as `int`.

3.3.7 Bitwise shift operators

A right shift of a negative-valued signed integral type copies the sign bit in.

3.3.8 Relational operators

Pointers may be compared using a relational operator even if they do not point to the same aggregate or union. (Conforming Extension)

3.5.1 Storage-class specifiers

The order in which registers are allocated is unspecified.

Structs and unions are never placed in a register, even if they would fit into one.

3.5.2.1 Structure and union specifiers

A bit-field may be declared with any integral type. The size of the declared type determines the "word" size for that bit-field, so a "word" may be 8, 16, or 32 bits wide.

A sequence of bit-fields with the same word size are packed into a word. No bit-field may be wider than its word size. If a bit-field would straddle a word boundary, it is placed in the next word.

Bit-fields are assigned beginning with the high-order bit of a word. An unnamed field with a width of 0 "closes out" the current word. A bit-field with a different word size from the preceding bit-field causes this closing out to happen automatically, just as a non-bit-field member does. (Conforming Extension)

A plain `int` bit-field is treated as a `signed int`.

All structures and unions are padded if necessary to be even-sized. Padding is inserted after a member only to meet the alignment requirements of the next member or at the end of the structure or union. Only odd-sized data items (for example, `char`, odd-sized array of `char`, `char`-size `enum` types, and so on) do not need to be aligned.

3.5.2.2 Enumeration specifiers

Enumerated types are of type `int` if the enums are always ints option is on. Otherwise, the type of an enumerated type is the first one in Table A-3 that is sufficient to hold all the values:

```
char
short
int
```

Table A-3 The type used for an enumerated type

3.5.3 Type qualifiers

In a compound assignment to a volatile-qualified type, Symantec C may generate a fetch and a store. Otherwise, each use of the volatile-qualified type results in a fetch or a store. If the volatile object is larger than 4 bytes, the compiler may need to generate multiple fetches or multiple stores.

3.5.4 Declarators

There is no limit to the number of declarators in an identifier.

3.6.4.2 The switch statement

The maximum number of case values in `switch` statements is limited only by available memory.

3.8.1 Conditional inclusion

The token `defined` has special meaning only if it appears literally with an `#if` or `#elif` directive. (Conforming Extension)

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such character constants can have a negative value.

3.8.2 Source file inclusion

The mechanism that Symantec C uses to locate header files is described in "How Symantec Compilers Look for Header Files" in Chapter 4.

The name in the `#include` directive is interpreted as a Macintosh file name. Case is not significant, and colons are treated as volume and directory separators.

3.8.3 Macro replacement

A macro argument that consists of no preprocessing tokens is treated as no tokens. (Conforming Extension)

3.8.6 Pragma directives

See "#pragma Directives" in Chapter 4 for a complete list and description of all the `#pragma` directives.

3.8.8 Predefined macro names

The following predefined macro names cannot be undefined or redefined: `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`. (Conforming Extension)

The identifier `defined` is interpreted specially only in the context of `#if` or `#elif` directives. It may not be undefined or redefined. (Conforming Extension)

`__DATE__` and `__TIME__` are always available.

4.1.5 Common definitions <stddef.h>

The following types are defined in `<stddef.h>`

Type	Definition
`NULL`	`((void *) 0)`
`size_t`	`unsigned int`
`ptrdiff_t`	`int`

Table A-4 Common types defined in stddef.h

4.2 Diagnostics <assert.h>

The diagnostic printed by assert is of the form "Assertion failed: *expression*, file *xyz*, line *nnn*". The message appears on `stderr`, and, if possible, is followed by the message "press return to exit".

4.3.1 Character-testing functions

The character-testing functions operate on the ranges in Table A-5.

Function	Range
`isalnum()`	`a-z, A-Z, 0-9`
`isalpha()`	`a-z, A-Z`
`iscntrl()`	`0x00-0x1F, 0x7F`
`islower()`	`a-z`
`isprint()`	`0x20-0x7E`
`isupper()`	`A-Z`

Table A-5 Character-testing functions

4.5.1 Treatment of error conditions

The values returned by mathematics functions on domain errors are documented for each function in the online Standard Libraries Reference.

The mathematics functions do not set `errno` to `ERANGE` on underflow range errors.

4.5.6.4 The fmod function

If the second argument to `fmod` is `0`, `fmod` returns `0` and sets `errno` to `EDOM`.

4.7.1.1 The signal function

The set of signals, their semantics, and the default handling of signals are described in the documentation of `signal` in the online Standard Libraries Reference.

No implementation-defined blocking of signals is performed. The default handler is reset if a `SIGILL` signal is received.

4.9.2 Streams

The last line of a text stream does not require a terminating newline character. Trailing blanks are not stripped off when writing to a text stream. No null characters are appended to a binary stream.

4.9.3 Files

The file position indicator of an append mode stream initially points to the end of the file. Zero-length files actually exist. A write on a text stream does not truncate beyond that point. Writing in place is allowed.

Symantec C implements buffered and unbuffered I/O. Line-buffered I/O is the same as fully buffered I/O.

The rules for composing valid file names are identical to Macintosh file-naming rules. In general, file names may have up to 31 characters, and the `:` character separates directory names. See the online Standard Libraries Reference for more information.

The same file can be opened more than once if the Macintosh file system allows it.

4.9.4.1 The remove function

It is an error to remove an open file.

4.9.4.2 The rename function

It is an error to use `rename` to give a file a name that already exists.

4.9.5.2 The fflush function

If you use `fflush` on a stream that is being used for input or update, and the most recent operation was input, the buffer is cleared and the next read will go to disk. (Conforming Extension)

The `fflush` function always returns the buffer to a neutral state, so this function can be used to switch between input and output, regardless of the last I/O operation. (Conforming Extension)

4.9.6.1 The fprintf function

If the conversion specification for the `fprintf` function contains a `#` flag and the conversion specifier is `s`, the matching argument is treated as a Pascal string. (Conforming Extension)

The output for the %p conversion in fprintf is the same as %.8lX, eight digit, hexadecimal integer that uses capital letters and leading zeros.

4.9.6.2 The fscanf function

The input format for the %p conversion in fscanf is the same as %lx, a hexadecimal integer.

A dash (-) character that is neither the first nor the last character in the scanlist for %[conversion is treated as a range specifier. For example, %[0-9] means the range of characters from '0' to '9' inclusive.

4.9.9.1 The fgetpos function

If fgetpos fails, errno is set to ENODEV.

4.9.9.4 The ftell function

If ftell fails, errno is set to ENODEV.

4.9.10.4 The perror function

The function perror prints the supplied error string, the string "Error: ", and the value of errno.

4.10.3 Memory management functions

If the number of bytes of memory requested from calloc, malloc, or realloc is 0, these functions return a pointer to a unique zero-size block of memory.

4.10.4.1 The abort function

The abort function closes all open and temporary files.

4.10.4.3 The exit function

The argument to the exit function is always ignored.

4.10.4.4 The getenv function

The getenv function always returns NULL.

4.10.4.5 The system function

The system function always ignores its argument and always returns 0 (zero).

4.11.6.2 The strerror function

The string that strerror returns is "Error: " and the value of its argument.

4.12.1 Components of time

The time functions are not aware of daylight savings time or the local time zone.

4.12.2.1 The clock function

The `clock` function returns the number of ticks (60ths of a second) since the Macintosh was turned on.

Symantec C Extensions

This section describes the extensions to ANSI C that are enabled when you turn off ANSI conformance.

pascal keyword

The identifier `pascal` is reserved to define functions that follow Pascal calling conventions. For examples, see "Calling Toolbox Routines" in Chapter 3.

C++ style comments

Two slashes (`//`) introduce a comment. The comment ends at the newline character.

Identifiers after #else and #endif

An identifier after an `#endif` or `#else` preprocessing directive is ignored.

```
#ifdef DEBUG
   printf( "oops!\n");
#endif DEBUG
```

Function prototypes

The parameter list `(...)` is allowed in a function prototype.

Dimensionless arrays allowed

Dimensionless arrays are allowed as the last member of `struct` definitions. The member does not contribute to the size of the `struct`. For example:

```
struct {
   short count;
   char data[];
} CountData;    /* sizeof(CountData) is 2 */
```

void *

The type `void *` is compatible with function pointers and data pointers.

Predefined symbols

The preprocessor symbol `SYMANTEC_C` is defined as the hex version number of Symantec C++. The current version is 0x800. The preprocessor symbol `THINK_C` is defined as `6` in THINK C 6.0 and 6.5, and as `5` in THINK C 5.0. It is defined as `1` in THINK C 4.0. It is not defined at all for prior versions.

Part II - Symantec C++ Language Reference

Introduction

This section describes how Symantec C++ implements those aspects of the C++ language that are denoted as implementation-defined in *The Annotated C++ Reference Manual.*

This chapter makes use of two abbreviations: ARM and Gray. ARM refers to *The Annotated C++ Reference Manual,* by Margaret Ellis and Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1990. Gray refers to *The C++ Programming Language, Second Edition* by Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1991.

This section is organized by the numbered sections of the ARM, with individual subsections marked with applicable references in both the ARM and Gray books. The ANSI committee is using the ARM as the basis for its definition of the C++ language; this section, however, includes references to both the ARM and Gray books.

Page numbers refer to the place in the specified reference where the implementation-dependent behavior is noted, not necessarily to the beginning of the section.

Note

This chapter identifies the differences between Symantec C++ and a standard C++. It does not define C++ language, nor does it explain C++ to a new user. See the ARM for a definition of C++ language, and the Gray book for C++ instruction.

When the ANSI C++ standard becomes publicly available, we will compare our Symantec C++ implementation to the ANSI standard.

Lexical Conventions

The numbers in this section correspond to the sections in the Annotated C++ Reference Manual (ARM).

§2.3 Identifiers

[ARM p. 6, Gray p. 478]

Identifiers in Symantec C++ have a maximum size of 1024 characters. They may have upper- and lowercase letters, numbers, or underscores (_). The underscore counts as a letter. All characters are significant. Identifiers are case-sensitive and must begin with a letter.

§2.5.2 Character Constants

[ARM p. 10, Gray pp. 480-1]

The mapping of characters in the source character set to the execution character set is one-to-one. The basic execution character set consists of all 256 Macintosh characters. You can represent all integer character constants or escape sequences with the basic execution character set.

Multi-character constants are type `int` and can contain between one and four characters from the execution character set. If the constant has more than four characters, then the compiler generates an error. If a character string of three or four characters is assigned to a `short`, then the last two characters are used in the assignment. For example, the following statement assigns `CD` (`0x4344`) to `foo`.

```
short foo = 'ABCD';
```

If the character following the backslash character is not one of the defined escape sequences, then the compiler generates an undefined escape sequence error.

§2.5.4 String Literals

[ARM pp. 10–11, Gray p. 482]

If a string literal begins with the sequence `\p` or `\P`, the compiler treats it as a Pascal string. The compiler replaces the `\p` or `\P` with the length of the string. Null (`'\0'`) is not appended to Pascal strings. Pascal strings, therefore, are restricted to 255 characters. Longer strings are truncated.

You can modify a string literal, but try to avoid doing so. If you modify a string literal, you may overwrite other global values.

The type of `wchar_t` is defined as a `short int` in `stddef.h`.

Basic Concepts

§3.4 Start and Termination

[ARM p. 19, Gray p. 485]

Every C++ program must contain a function called `main()`. Its default type is `int`, and it has external linkage. You can take the address of `main()`.

§3.6.1 Fundamental Types

[ARM p. 22 (cf. p. 7; 3.2.1c), Gray pp. 486–7]
The compiler allocates types on word boundaries. Within structures, you can set alignment on byte, word (2 bytes), or long word (4 bytes, the default) boundaries.

The compiler treats a `char` object that is not declared either `signed` or `unsigned` as a `signed` value.

Integers are represented as two's complement binary numbers. Table A-6 lists the sizes of the integer types.

Type	Bytes
`char`	1
`short`	2
`int`	4
`long`	4

Table A-6 The size of integer types

The header `limits.h` specifies the largest and smallest values of the integral types. Table A-7 defines the limits of the integral types.

Variable	Value	Definition
`CHAR_BIT`	8	Maximum bits in a byte
`SCHAR_MAX`	+127	Maximum value of `signed char`
`SCHAR_MIN`	-128	Minimum value of `signed char`
`UCHAR_MAX`	255	Maximum value of `unsigned char`
`CHAR_MAX`	`UCHAR_MAX` `SCHAR_MAX`	Maximum value of `char`
`CHAR_MIN`	0 `SCHAR_MIN`	Minimum value of `char`

Table A-7 Limits of integral types

Variable	Value	Definition
SHRT_MAX	+32767	Maximum value of `short`
SHRT_MIN	-32768	Minimum value of `short`
USHRT_MAX	65535	Maximum value of `unsigned short`
LONG_MAX	+2147483647	Maximum value of `long`
LONG_MIN	-2147483648	Minimum value of `long`
ULONG_MAX	4294967295	Maximum value of `unsigned long`
INT_MAX	LONG_MAX	Maximum value of `int`
INT_MIN	LONG_MIN	Minimum value of `int`
UINT_MAX	ULONG_MAX	Maximum value of `unsigned int`

Table A-7 Limits of integral types *(Continued)*

Table A-8 shows the two ways you can represent floating-point values.

Type	Default Size in Bytes	Format
`float`	4	IEEE single precision
`double`	8	IEEE double precision
`long double`	8	IEEE 8 byte

Table A-8 Floating-point values

You can find these formats documented in detail in the *Inside Macintosh PowerPC Numerics Manual* (Addison-Wesley) by Apple Computer, and the *MC68881/682 User's Manual* (Motorola).

The header `floating.h` defines the characteristics of the floating types. Table A-9 summarizes the floating types.

Variable	Value	Definition
FLT_DIG	7	Decimal digits of precision
FLT_MANT_DIG	24	Number of base FLT_RADIX digits in mantissa
FLT_MAX_10_EXP	38	Maximum positive integer *n* such that 10 raised to the *n*th is within the range of normalized floating-point numbers
FLT_MAX_EXP	128	Maximum positive integer *n* such that FLT_RADIX raised to the *n*th minus 1 is representable
FLT_MIN_10_EXP	-37	Minimum negative integer *n* such that 10 raised to the *n*th is within the range of normalized floating-point numbers
FLT_MIN_EXP	-125	Minimum negative integer *n* such that FLT_RADIX raised to the *n*th minus 1 is a normalized floating-point number
FLT_RADIX	2	Radix of exponent representation
FLT_ROUNDS	1	Direction of rounding
FLT_MAX	3.402823e+38	Maximum representable floating-point number

Table A-9 Floating-point limits

Variable	Value	Definition
FLT_MIN	1.175494e-38	Minimum normalized positive floating-point number
FLT_EPSILON	1.192093e-7	Minimum positive number *x* such that 1.0+*x* does not equal 1.0
DBL_DIG	15	Decimal digits of precision
DBL_MANT_DIG	53	Number of base FLT_RADIX digits in mantissa
DBL_MAX_10_EXP	308	Maximum integer *n* such that 10 raised to the *n*th is representable
DBL_MAX_EXP	1024	Maximum integer *n* such that FLT_RADIX raised to the *n*th minus 1 is representable
DBL_MIN_10_EXP	-307	Minimum negative integer *n* such that 10 raised to the *n*th is within the range of normalized floating-point numbers
DBL_MIN_EXP	-1021	Minimum negative integer *n* such that FLT_RADIX raised to the *n*th minus 1 is a normalized floating-point number
DBL_MAX	1.797693e+308	Maximum representable floating-point number
DBL_MIN	2.225074e-308	Minimum normalized positive floating-point number

Table A-9 Floating-point limits *(Continued)*

Variable	Value	Definition
DBL_EPSILON	2.220446e-16	Minimum positive number *x* such that 1.0+*x* does not equal 1.0
LDBL_DIG	15	Decimal digits of precision
LDBL_MANT_DIG	53	Number of base FLT_RADIX digits in mantissa
LDBL_MAX_10_EXP	308	Maximum integer *n* such that 10 raised to the *n*th is representable
LDBL_MAX_EXP	1024	Maximum integer *n* such that FLT_RADIX raised to the *n*th minus 1 is representable
LDBL_MIN_10_EXP	-307	Minimum negative integer *n* such that 10 raised to the *n*th is within the range of normalized floating-point numbers
LDBL_MIN_EXP	-1021	Minimum negative integer *n* such that FLT_RADIX raised to the *n*th minus 1 is a normalized floating-point number
LDBL_MAX	1.79693e+308	Maximum representable floating-point number
LDBL_MIN	2.225074e-308	Minimum normalized positive floating-point number
LDBL_EPSILON	2.220446e-16	Minimum positive number *x* such that 1.0+*x* does not equal 1.0

Table A-9 Floating-point limits *(Continued)*

Standard Conversions

§4.1 Integral Promotions

[ARM pp. 31–2, 322, Gray p. 489]

Symantec C++ follows ANSI C in that integral promotion is "value-preserving." See ARM p. 32 for an in-depth discussion of this and its relationship to older C++ implementations.

§4.2 Integral Conversions

[ARM p. 33, Gray p. 489]

When the compiler demotes an integer to a smaller, signed integer, the compiler copies the low-order bits; the high-order bit of the smaller integer becomes the sign bit.

When the compiler tries to convert an unsigned value to a signed integer of equal length, but the compiler cannot represent the unsigned value by the signed type, then the representation bit pattern doesn't change. The high-order bit, which in the unsigned interpretation contributes to the value, is now interpreted as the sign bit.

§4.3 Float and Double

[ARM p. 33, Gray p. 489]

If you convert a floating-point value that is in a range that the compiler can represent but not exactly (such as 0.1, which becomes a repeating binary fraction), the compiler rounds the result according to the rounding mode. The default rounding mode is to round to the nearest, and you can change this mode by editing the value `FLT_ROUNDS` in `floating.h`, which is documented in Table A-9.

§4.4 Floating and Integral

[ARM pp. 33–4, Gray p. 489]

If you convert an integral type to a floating-point type, and that value is in the range the compiler can represent but not exactly, the compiler rounds the result according to the current rounding mode. The default rounding mode is to round to the nearest integer, and you can change it.

§5.0 Expressions

[ARM p. 46 (cf. p. 72, §5.6), Gray p. 492]

The compiler ignores integer overflows.

Symantec C++ handles division by zero in several different ways, depending on context.

If you try to divide this by zero...	The compiler returns...
A constant expression	An error
Any number during constant folding	An error
An integer	A microprocessor exception (System Error #4)
A floating-point number	INF (+∞)

Table A-10 Division by zero

For further information, consult the *Inside Macintosh PowerPC Numerics Manual.*

§5.2.4 Class Member Access

[ARM p. 53]

The compiler doesn't convert values stored in a member of a union and then accessed through another member. For example:

```
union u_tag {
   int ival;
   float fval;
} u_obj;
int i;
u_obj.fval = 4.0;
i = u_obj.ival;
```

assigns `0x40800000` to `i`.

§5.3.2 Sizeof

[ARM p. 58, Gray p. 497]

The type `size_t` is defined as an `unsigned int` in `stddef.h`.

§5.3.3 New

[ARM p. 61, Gray p. 499]

Allocation is performed inside an object's constructor if one is present.

§5.4 Explicit Type Conversion

[ARM p. 71, 37, Gray pp. 500–2]

You can convert a pointer to an integral type large enough to hold it (that is, 4 bytes) with no changes, though the compiler interprets the bit pattern as the integral type. When converting to a smaller integral type, the compiler uses the low-order bytes of the pointer.

If you convert an integral type to a pointer, the compiler promotes and sign-extends smaller integral types to the appropriate integral type without losing information.

You can cast away the "`const`"-ness of an object, so that it is possible to modify the value of the constant object. If a pointer or reference to a `const` is cast to a pointer or reference to non-`const`, writing to the pointer or reference succeeds if the original pointer or reference contained a valid address.

§5.6 Multiplicative Operators

[ARM p. 72, Gray p. 503]

When two integers are divided with the / operator, where the result is inexact and one and only one of the operands is negative, the result is the smallest integer greater than the algebraic quotient (such as `-23/4 = -5`). If the % operator is used, where the division is inexact and one and only one of the operands is negative, the result is negative (such as `-23%4` is equal to `-3`). If the right operand of the `%` or / operator is 0, then the compiler signals a microprocessor exception.

§5.7 Additive Operators

[ARM p. 73, Gray p. 503]

The compiler treats memory as a linear address space. You can reference out of the bounds of an array without the compiler detecting it. For example:

```
int a[10];
void f()
{
    int* p = &a[10];
    *p = 0xdeadbeef;
}
```

You can subtract two pointers to objects in the same array to find the number of elements separating the operands. The result is of type `ptrdiff_t`, defined as `long` in `<stddef.h>`. Subtracting pointers of differing types results in an error, though explicit casting lets you do the operations.

§5.8 Shift Operators

[ARM p. 74, Gray p. 504]

If the right operand of the left-shift operator << is negative, then the result is undefined. If it is greater than or equal to the length in bits of the promoted left operand, then it is taken modulo 64 and used, with the usual result that all the bits are shifted out of the left operand.

When the left operand of the >> operator is a signed type and negative, the compiler performs a signed right shift.

Declarations

§7.1.6 Type Specifiers

[ARM p. 110, Gray p. 521]

A declaration with the specifier `volatile` tells the compiler that the declared object can change in an undetectable way. These objects are not optimized.

§7.2 Enumeration Declarations

[ARM pp. 114–5, Gray p. 523]

The size of an enumeration is the largest integral type that holds the largest value in the enumeration, unless the enums are always ints option is on (See "enums are always ints" in Chapter 4 for more information). You can cast to an enumeration, but you may not get the results you expected. For example:

```
enum color {red, yellow, green=20, blue};
color c3 = color(600);
int i = c3;
```

Here, `i` will receive 58 since each enumerator is stored as a single byte. Changing the enumeration to:

```
enum color {red, yellow, green = 2000, blue};
```

allocates each enumerator as a `long`, into which 600 fits.

§7.3 Asm Declarations

[ARM p. 115, Gray p. 524]

An `asm` declaration lets you embed short assembly language fragments into the body of your C++ code. It takes a variable number of integer arguments representing the machine language instructions. The compiler then inserts these instructions into the generated code. For example, the declaration:

```
asm (0x700A,0x5A80,0x2600);
```

inserts these instructions into the code:

```
MOVEQ     #$0A,D0
ADDQ.L    #$05,D0
MOVE.L    D0,D3
```

§7.4 Linkage Specifications

[ARM p. 116, 118, Gray p. 524]

Symantec C++ supports C, C++, and Pascal linkage types.

Classes

§9.2 Class Members

[ARM p. 173, 241, Gray p. 545]

The compiler allocates non-static data members of a class in order of appearance in the source file, regardless of intervening access specifiers.

§9.6 Bit-Fields

[ARM pp. 184–5, Gray p. 550]

You can declare a bit-field with any integral type. The size of the declared type determines the "word" size for that bit-field, so a "word" may be 8, 16, or 32 bits wide.

A sequence of bit-fields with the same word size is packed into a word. No bit-field may be wider than its word size. If a bit-field would straddle a word boundary, the compiler places it in the next word. For example, the bit-field declaration

```
struct bits {
   int b1: 24;
   int b2: 8;
   int b3: 24;
   int b4: 24;
};
```

is represented in memory as:

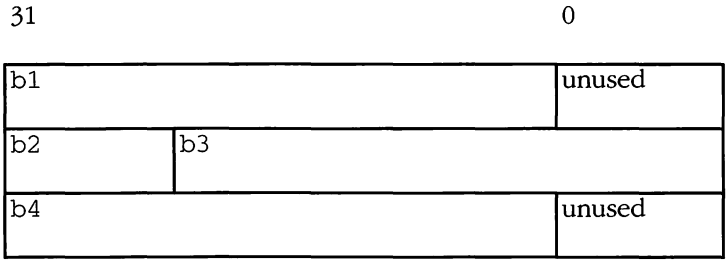

Figure A-2 Sample code as represented in memory

The compiler assigns bit-fields beginning with the high-order bit of a word. An unnamed field with a width of 0 "closes out" the current word. A bit-field with a different word size from the preceding bit-field causes this closing out to happen automatically, just as a non-bit-field member does.

The compiler treats a plain `int` bit-field as a signed `int`.

Special Member Functions

§12.2 Temporary Objects

[ARM pp. 267–8, Gray p. 572]

The compiler destroys temporary objects when their values go out of scope.

Templates

This section has undergone fairly substantial revision by the ANSI C++ committee. The Symantec compilers implement several rules from the latest ANSI C++ draft that did not exist in the ARM. Changes below taken from the latest ANSI C++ draft standard (dated September 20, 1994) are specifically cited as coming from this document.

§14.1 Templates

By default, the Symantec compilers give local linkage to template classes and functions. This renders certain well-formed template programs erroneous and places limitations on what can be done

with templates without using the `template_access` pragma. For example:

```
template <class T>
void f(T &t);

void main(void)
{
   int i;
   f(i);
}
```

In the ARM, this program would be well-formed because the linkage of `f` would be global. If the definition for `f` appeared in a different translation unit that caused the expansion of `f(int)`, the program would link and run properly. Under the Symantec compilers this program would be ill-formed because function `f` has local linkage and is not defined within the translation unit.

The error reported would be:

```
Error: no definition for static 'f(int&)'
```

If the definition of `f` is found in another translation unit, the `#pragma template_access` directive must be used to inform the compiler that the definition for `f` will be seen elsewhere. As in:

```
#pragma template_access extern
                        // Templates are
                        // defined elsewhere
template <class T>
void f(T &t);

void main(void)
{
   int i;
   f(i);
}
```

The file containing the definition for `f(T)` would be:

```
#pragma template_access public// I am
                    //expanding templates here
template <class T>
void f(T &t)
{
   ...
}

#pragma template f(int)
```

See "#pragma [SC] template_access" in Chapter 4 for more information on the template_access pragma.

Due to their local linkage by default, template classes that contain static data will not behave properly. For example:

```
file f.h:

template <class T> class X {
static int i;
};
template <class T>
int X<T>::i = 10;

file f1.c:
#include <f.h>

void f1(void)
{
   X<int>::i = 5;
}

file f2.c:
#include <f.h>

void f2(void)
{
   X<int>::i = 6;
}
```

Due to the local linkage of template classes, `f1.c` and `f2.c` each have their own independent copy of `X<int>::i`. The compiler will not diagnose this is an error, but the program may not behave as the user intends. In order to get the proper linkage for template class static data, the `#pragma template_access` directive must be

used to control the linkage of the template classes involved. For example:

```
file f.h:

#pragma template_access extern
template <class T> class X {
static int i;
} ;
template <class T>
int X<T>::i = 10;

file f1.c:
#include <f.h>

void f1(void)
{
   X<int>::i = 5;
}

file f2.c:
#include <f.h>
#pragma template_access public

void f2(void)
{
   X<int>::i = 6;
}

#pragma template X<int> // Provides definition
                        // for X<int>::i=10
```

§14.4 Function Templates

The Symantec compiler implements the following rules from the ANSI C++ draft specification section 14.9.3 [temp.over]:

1. Look for an exact match (13.2) on functions; if found, call it.

2. Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

3. Look for a match with conversions. For arguments to ordinary functions and for arguments to a template function that corresponds to parameters whose type does not depend on a deduced *template-parameter*, the ordinary best-match rules apply. For template functions, only the following conversions listed below apply. After

the best matches are found for individual arguments, the intersection rule (_over.match.args_) is used to look for a best match; if found call it.

For arguments that correspond to parameters whose type depends on a deduced template parameter, the following conversions are allowed:

- For a parameter of the form B<params>, where params is a template parameter list containing one or more deduced parameters, an argument of type "class derived from B<params>" can be converted to B<params>. Additionally, for a parameter of the form B<params>*, an argument of type "pointer to class derived from B<params>" can be converted to B<params>*. Similarly for references. Also, for a parameter of the form T, an argument of type "T B::* where B is a base of D<params>" can be converted to T D<params>::*.

- A pointer or reference can be converted to a more qualified pointer (reference) type, according to the rules in 4.10 (_conv.ref_).

- "array of T" to "pointer to T".

- "function ..." to "pointer to function...".

In each case, if there is more than one alternative in the first step (1, 2, 3) that finds a match, the call is ill-formed.

This rule allows:

```
template <class T>
double f(T &t, double d );

void f(void)
{
   double d;
   int i = 0;

   d = f( i, i );     // Allowed by rule #3
                      // in ANSI,
                      // ill-formed in the ARM
   }
```

which previously would have been ill-formed.

Another modification was to the behavior of template specialization in the ARM, declaring a function that could be expanded from a template was defined as causing a new expansion unless the new declaration contained a definition. For example:

```
template <class T>
void f(T t);
{
   ...
}

void f(int);        // Created an expansion of
                    // f(T t) to satisfy f(int)

void f(double d)    // Ambiguous in the ARM,
{                   // either error or a valid
                    // specialization of f(T)
}
```

In the ANSI draft there is new syntax for explicitly instantiating a template and the declaration `void f(int)` will never generate an expansion of the template function `f`. The current compiler does not implement the new syntax for explicit specialization, but does implement pragmas that accomplish this for the user. The compiler, therefore, never allows the declaration of `f(int)` to instantiate the template function. The previous code example will report an undefined symbol for `f(int)`, treating it as a specialization of the template function `f`. In order to get the same behavior in the Symantec compiler, use:

```
template <class T>
void f(T t);
{
   ...
}

#pragma template f(int)  // Create an
                         // expansion of
                         // f(T t), namely
                         // f(int)

void f(double d)       // valid specialization
                       // of f(T)
{
}
```

§14.7 Friends

The Symantec compilers implement the following restriction regarding template expansion and name injection from the ANSI C++ draft specification, section 14.2.4 [temp.inject]:

"Names that are not template members can be declared within a template class or function. However, such declarations must match declarations in scope at the point of their declaration or instantiation."

This means that many templates that presently declare friend operators or other template friend functions must be provided with declarations prior to the instantiation of the template class. For example:

```
template <class T> class X {
    ...
    friend ostream &operator<<(ostream& s, X<T>& m);
} ;

X<int> xi;      // With the ARM, this is OK, with
ANSI,
                // this is ill-formed
                // Will receive:
                // Error:   Non-local name 'operator
                // <<(ostream&,X<int>&)' cannot be
                // declared in a template
                // instantiation.
```

This error will only be reported if ANSI conformance (relaxed or strict) is enabled. In order to avoid the error, simply declare the friend prior to the first instantiation of the template class, as in:

```
template <class T> class X {
    ...
    friend ostream &operator<<(ostream& s, X<T>& m);
} ;
template <class T>
ostream & operator <<(ostream & s, X<T>& m );

X<int> xi;      // This is OK in both the ARM and
                // ANSI specifications.
```

Note

The class body for a class is parsed and this error is produced regardless of any `#pragma template_access` options specified. The `#pragma template_access` directive controls the generation of code and data, the declarations of class members, and other symbols are always expanded regardless of the `#pragma template_access` setting. See "#pragma [SC] template_access" in Chapter 4 for more information on the template_access pragma.

Exceptions

§15 Exception Handling

The Symantec compilers do not currently implement any of the exception handling portion of the ARM or ANSI specifications. The keywords `throw`, `try`, and `catch` are reserved in this release.

Preprocessing

§16.4 File Inclusion

[ARM pp. 375–6 (cf. 16.3.2c), Gray pp. 610–11]

These are the rules Symantec C++ uses to find header files:

#include statement	**Symantec C++**
`<filename.h>`	Look first in the referencing folder, then in the system tree.
`"filename.h"`	Looks first in the referencing folder, then in the project tree, and finally in the system tree.

The referencing folder is the one that contains the file that has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h`, and that file in turn has the line `#include "MyUtilTypes.h"`, Symantec C++ looks for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

§16.5 Conditional Compilation

[ARM p. 377, Gray p. 613]

No limit has been placed on the number of `#if` directives that you can nest.

§16.8 Pragmas

[ARM p. 378, Gray p. 613]

Symantec C++ defines many pragmas. The preprocessor produces a warning for unrecognized pragmas. See "#pragma Directives" in Chapter 4.

§16.10 Predefined Names

[ARM p. 379, Gray p. 614]

Symantec C++ does not define the predefined name `__STDC__` .

Error Messages B

This appendix lists and describes error and warning messages generated by Symantec C, Symantec C++, and Symantec Rez. Error messages generated by the Symantec Debugger and the Symantec Internal Linker are described in the *Symantec C++ User's Guide and Reference.*

Use this reference to:

- Check or confirm that an error has been reported.
- Discover possible causes for an error.
- Discover possible ways to correct an error.

All the error messages in this Appendix indicate in which category (C/C++, C, C++, Symantec Rez) they belong. For example, *C/C++* means that either compiler can generate the message, while *C++* means that only the PowerPC C++ compiler can produce the message. Messages marked *Warning* indicate code that does compile but that may not execute as you expect. This appendix lists messages in the order of: messages that begin with symbols, followed by messages with variable first words, followed by the remaining alphabetized messages.

Some descriptions contain a margin note that refers to sections in *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1990. These sections contain information that will help fix your program.

Contents

Recognizing Compiler Error Messages

When the compiler encounters a line in source code it can't compile or believes is incorrect, it usually prints the reason in the Build Errors window, as in Figure B-1 below, and continues compiling your project.

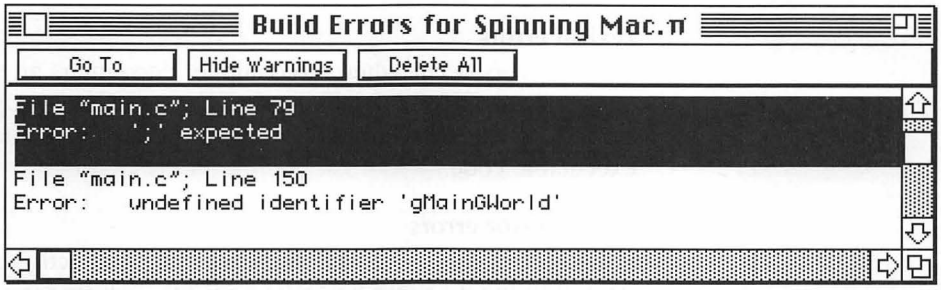

Figure B-1 Build Errors window

To see the line described in the message, double-click the message, or click Go To at the top of the window. If the error is in a source file, the Symantec Editor opens the file and selects the line, as in Figure B-2 below. The Symantec Editor keeps track of your edits.

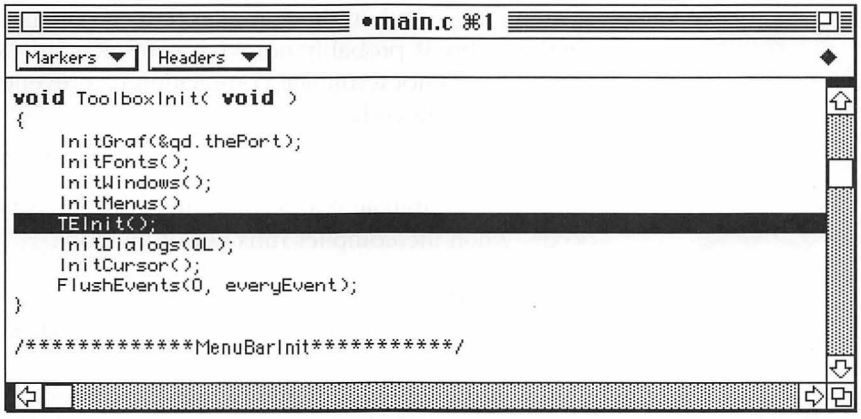

Figure B-2 Symantec Editor with incorrect line of code highlighted

Error Message Types

There are six error message types. Each message usually contains specific information about the problem. The compiler normally lists four errors of the preprocessor, syntax, or lexical types before exiting. Use the Report all errors in a file option to let compilation continue to the end of the source file before exiting with an error.

Lexical errors

Lexical errors occur when the compiler encounters an unidentified or incomplete token. While they do not terminate compilation immediately, lexical errors do prevent the compiler from generating executable code.

Preprocessor errors

Errors can occur in one of the preprocessing directives. While they do not terminate compilation immediately, preprocessor errors can prevent the compiler from generating executable code.

Syntax errors

While they do not terminate compilation immediately, syntax errors can prevent the compiler from generating executable code.

Warnings

Warnings occur when the compiler finds a statement that is legitimate but is probably not what you intended. Warnings are not errors and do not terminate compilation or prevent the compiler from generating code.

Fatal errors

Fatal errors terminate compilation immediately. A typical fatal error occurs when the compiler runs out of memory.

Internal errors

Internal errors, a class of fatal error, take the following form:

```
file/line #
```

An assertion failure within the compiler generates this type of error. The error number is useful only in designating where the error occurs in the compiler code. The cause of this message may be an error in source code that the compiler cannot handle intelligently or a bug in the compiler itself. If your code generates this type of error,

report it to Symantec, even if your code causes the error. With this information, Symantec can improve error reporting in future releases.

How to report an internal error

Before reporting an internal error to technical support, try to isolate the error in a small program fragment. Use the following procedure:

1. Place all included code into the main program body using the **Preprocess** command on the **Build** menu.
2. Find the approximate cause of the error by backtracking and removing excess code to isolate a short program that demonstrates the fault.
3. Use mnemonic names for objects and variables in the sample code. Code containing `class Base` rather than `class Hyperxytrisms59` is much easier for the technical support staff to understand.
4. If applicable, put the offending code in an `#ifdef BUG .. #endif` block.
5. Write a comment header with the following information: your name, telephone number, address, version of the compiler, and the Symantec Project Manager as well as any other software involved, the nature of the problem, and other relevant details.

A short bug report lets the technical support staff quickly find and reproduce the problem.

Symantec C++ for Power Macintosh Error Messages

This list contains error messages that the Symantec compilers may generate:

 # must be followed by a parameter

(C/C++) The number sign operator must appear in front of a macro parameter. For example, `#c`.

'#else' or '#elif' found without '#if'

(C/C++) More `#else` or `#elif` preprocessor directives appear than preceding `#if`, `#ifdef`, or `#ifndef` directives.

'#endif' found without '#if'
(C/C++) More `#endif` preprocessor directives appear than preceding `#if`, `#ifdef`, or `#ifndef` directives.

#printf statements can't have more than 20 arguments
(Symantec Rez) Symantec Rez won't compile any `#printf` statement with more than 20 arguments.

#printf statements do not end with a ';'
(Symantec Rez) Unlike C, in which the `printf` statement is a function call, Symantec Rez considers `#printf` a preprocessor statement, similar to a `#define`.

$$resource statement failed
(Symantec Rez) A `$$resource` statement fails if the source file can't be found, or if the given resource type, ID or name is invalid.

$$shell is not supported
(Symantec Rez) The Symantec Project Manager does not support exported shell variables, so Symantec Rez will not compile files that contain any `$$shell` statements.

'(' expected
(C/C++) The compiler expects the expression after the `if`, `while`, or `for` keywords to be enclosed in parentheses.

')' expected
(C/C++) The compiler expects a set of parentheses to be closed. Check for a pair of mismatched parentheses or a bad expression.

// comments are not ANSI C
(C/C++) You cannot use `//` comments with strict ANSI conformance in the C compiler.

':' expected
(C/C++) The compiler expects a colon after a constant expression in a `case` statement and after the keywords `public`, `private`, and `protected` in a class declaration.

'::' expected
(C++) If the compiler generates this message, let us know. The first validated entry to qa@bedford.symantec.com wins a prize of our choice.

'::' or '(' expected after class *'identifier'*

(C++) The compiler expects two colons or an open parenthesis after a class name in an expression. Casting, however, does not allow two colons. For example:

```
class x;
f=*(x*)y;
```

';' expected

(C/C++) The compiler expects a semicolon at the end of a statement.

'<' expected

See ARM 14.1 for more information.

(C++) In a class or function template, the argument list must be placed between angle brackets.

'=', ';' or ',' expected

(C/C++) A variable is declared incorrectly. A declaration must include an equals sign, a semicolon, or a comma after the variable name.

'>' expected

See ARM 14.1 for more information.

(C++) In a class or function template, the argument list must be placed between angle brackets.

']' expected

(C/C++) The compiler expects a close bracket at the end of an array declaration or reference.

'{' expected

(C/C++) The compiler expects an open brace.

'{' or tag identifier expected

(C/C++) The compiler expects a tag name or an open brace to follow the keywords `struct`, `class`, `union`, or `enum`.

'}' expected

(C/C++) The compiler expects a close brace.

***'identifier'* is a pure virtual function**

(C++) The compiler cannot directly call a pure virtual function.

'identifier' is a virtual base class of *'identifier'*

(C++) You cannot convert a pointer to a virtual base class into a pointer to a class derived from it. Also, you cannot create a pointer to a member of a virtual base class. For example:

```
class virtual_class {
public:
   int x;
};

class sub_class :
   virtual public virtual_class { };

void main()
{
   virtual_class *v;
   sub_class *s;
   int virtual_class::*i;

   s = (sub_class *) v;// error
   i = &sub_class::x;
}
```

'identifier' is already defined

(C/C++) You've already defined the item.

'identifier' is not a class template

(C++) The compiler expects to find the name of a class template but doesn't. If you are declaring a template member function, make sure the function's class name is a template. If you use a type of the form `foo<bar>`, make sure you declare as a template the class name before the less-than sign.

'identifier' is not a constructor

See ARM 12.6.2 for more information.

(C++) You can use a member initialization list only when you're defining base constructors and member initializers. For example:

```
struct base   { base(int);  };
struct other  { other(int); };

class sub : base {
   sub(int);       // A constructor.
   sub2(int);      // Just a method.
   other o;
} ;

sub::sub(int a) : o(a), base(a) { }// OK
sub::sub2(int a): o(a), base(a) { }// ERROR
```

'*identifier*' is not a correct struct, union or enum tag identifier

(C/C++) The `struct`, `union`, or `enum` tag identifier includes invalid characters or is already defined.

'*identifier*' is not a member of forward referenced struct '*identifier*'

(C/C++) You cannot access members of a `struct` if it has not been declared. For example:

```
struct X;
void f(x*Px) {
    Px->i = 0;      //error
}
```

'*identifier*' is not a member of struct '*identifier*'

See ARM 11.4 for more information.

(C/C++) The member *identifier* is not a member of this `class`, `struct`, or `union`. Make sure to spell the member name correctly and verify that the member actually belongs to the `struct` with which you're using it. If the member belongs to a different `struct` but you want to use it with this `struct` anyway, cast the `struct`. Also check for a class member function that is forward-referenced. For example:

```
class X;                // Forward reference
class Y {               // Declaration
  void g();
  /* . . . */
};

class Z {
  friend void X::f();// ERROR
  friend void Y::g();// OK
};
```

'*identifier*' is not a static variable

(C++) A static variable is not used as an argument to a static constructor when required.

'*identifier*' is not a struct or a class

(C++) You can derive new classes only from a `class` or a `struct`. It is not possible, for instance, to derive a class from a `union`.

'*identifier*' is not a variable

(C) You cannot use a `typedef` as a variable. For example:

```
typedef int FOO;
void f(void)
{
   int i;
   i = FOO;
}
```

'*identifier*' is not in function parameter list

(C/C++) The parameter *identifier* is not listed as a parameter to the function in the function definition.

'*identifier*' must be a base class

(C++) When naming a member of a base class in a derived class declaration, qualify the member with a base class identifier. For example:

```
class other;
class base {
private:
   /* . . . */
};

class sub : base {
public:
   other::a;   // ERROR: other must be a
   /* . . . */ //        base class of sub.
};
```

'*identifier*' must be a class name preceding '::'

(C++) The identifier before the double colon operator must be a `class`, a `struct`, or a `union`.

'identifier' must be a public base class

(C/C++) When you use the syntax `p->class::member`, *class* must be a public base class member of the class to which `p` is referring. For example:

```
class public_base {
public:
   int x;
};

class other_class {
public:
   int z;
};

class sub : public public_base {
   /* . . . */
};

void main()
{
   sub* s;
   s->public_base::x = 1;      // OK
   s->other_class::z = 1;      // ERROR
}
```

'identifier' previously declared as something else

(C/C++) You previously declared the identifier as another type. For example, you may have used a function without declaring it, so the compiler automatically declares it as a function returning an `int`. You cannot then declare that function to be something else.

'identifier' storage class is illegal in this context

(C/C++) Check for one of the following:

See ARM 14.1 for more information.

- You declared a template outside the global scope.
- You declared a function argument `static` or `extern`.
- You used an `auto` or `register` variable with global scope.

```
register int global;
         // ERROR: Can't declare global
         //        variable as register.

void f()
{
   template<class T> T ave(T* a, int size)
   {
         // ERROR: Can't declare template
         //        in a function.
   }
   /* . . . */
}
```

number* actual arguments expected for *'identifier'

(C/C++) The compiler expects a different number of arguments for the function or template. You may be using the function incorrectly, or you may be calling a function with a variable number of arguments without including its header file.

***number* exceeds maximum of *'number'* macro parameters**

(C/C++) A macro has more than the allowed number of macro parameters.

0 expected

(C++) A pure virtual function is declared incorrectly. The following is the correct syntax:

```
class X {
   virtual pure_virtual_func() = 0; // OK
   /* . . . */
}
```

0 or 1 expected

(C/C++/Symantec Rez) Only binary digits can follow the characters `0b`. No spaces should be placed between the `b` and the number.

a derived class member has the same name *'identifier'*

See ARM 11.3 for more information.

(C++) A base member's access cannot change when a derived class defines a member with the same name. For example:

```
class base {
public:
   int x, y;
   /* . . . */
};

class sub : base {
public:
   void x();
   base::x;     // ERROR: same name as x()
   base::y;     // OK
};
```

access declaration must be in public or protected section

See ARM 11.3 for more information.

(C++) A class member's access can change only if that class member is in a public or protected section. For example:

```
class base {
   int a;
public:
   int x;
};

class sub : private base {
   base::a;     // ERROR
public:
   base::x;     // OK: x is public
};
```

alignment must be 1, 2, 4

(C/C++) The value for the alignment in a `#pragma align` statement must be 1, 2, or 4.

already seen initializer for *'identifier'*

(C++) Either more than one member-initializer for the identifier exists, or more than one initializer for the base class exists. For example:

```
class base {
   int x;
   base(int);
};

class sub : base {
   base b;
   sub(int);
};

sub::sub(int a) : base(a+1),   // OK
                  b(a*2),      // OK
                  base(a-2)    // ERROR
{ x = a; }
```

ambiguous reference to base class *'identifier'*

(C++) This class has more than one base class, and it is not clear to which the program is referring.

ambiguous reference to function or member

(C++) This function or data member is ambiguous with an inheritance tree. For example:

```
class B {
   int i;
   void f();
};
class C : B {
};
class D : B,C
{
};
D d;
void g(void)
{
   d.i = 3;        //error
   d.f();          //error
}
```

ambiguous reference to function

(C++) In calling an overloaded function, more than one definition of the function matches the call. For example:

```
struct X {
   X(int);
};

struct Y {
   Y(int);
};

void f(X);  // f() can take an argument of
void f(Y);  // either type X or type Y.

void main()
{
   f(1);     // ERROR: Ambiguous,
             //        f(X(1)) or f(Y(1))?
   f(X(1)); // OK
   f(Y(1)); // OK
}
```

ambiguous type conversion

(C++) The compiler cannot find an unambiguous type conversion. For example:

```
struct X {
   operator int();
   operator void*();
};

void main()
{
   X x;

   if (x) ;           // ERROR
   if ((int) x) ;     // OK
   if ((void*) x) ;   // OK
}
```

argument of type *'identifier'* to copy constructor

See ARM 12.1 for more information.

(C++) Copy constructors for class X cannot take an argument of type X. Instead, use the reference to X.

argument to postfix ++ or -- must be int

See ARM 13.4.7 for more information.

(C++) Only declarations of the following form can declare overloaded functions for the prefix and postfix operators `++` and `--`:

```
operator ++()      // prefix ++X
operator ++(int)   // postfix X++
operator --()      // prefix --X
operator --(int)   // postfix X--
```

array dimension must be > 0

(C/C++) A negative number or zero cannot act as an array dimension when you declare an array.

array nesting too complex

(Symantec Rez) Symantec Rez will not compile a nested array that is more than 20 levels deep.

array of functions is illegal

(C) An array of pointers to functions, not an array of functions, can be declared. For example, instead of this:

```
int x[10](int, int);
      // ERROR: an array of functions
      //        returning int
```

use this:

```
int (* x[10])();
      // OK: an array of pointers to
      //     functions returning int
```

array of functions or refs is illegal

See ARM 8.4.3 for more information.

(C++) An array of pointers to functions, not an array of functions, can be declared. For example, instead of this:

```
int &x[10];
   // ERROR: Array of references to int
```

use this:

```
int *x[10];
   // OK: Array of pointers to int
```

An array of references is illegal in C++.

array or pointer required before '['
(C/C++) The brackets operator can only follow an array or pointer identifier.

assignment to 'this' is obsolete, use X::operator new/delete
(C++) Avoid performing storage management by assigning to *this*. Instead, overload the operators `new` and `delete`.

Warning
Assigning to *this* is not part of the latest definition of C++, and future compilers may not support it.

at least one parameter must be a class or a class&
(C++) An operator overloaded function that is not a class member must have at least one parameter that is a class or class reference.

attempt to initialize an array in a type statement
(Symantec Rez) The only kind of arrays that can be initialized when declared are `rect` and `point`. All other arrays must be defined as a resource statement.

bad constant value
(Symantec Rez) Some statements require a constant value. Make sure that your statement is not attempting to use a computed value when a constant value is required. For example, an enumerated value must be a constant value.

```
type 'test' {
   short x = unknown_label_value;
         /* must be a constant value */
};
```

bad member-initializer for *'identifier'*

(C++) A syntax error exists in the base class initializer for the class identifier. For example:

```
struct base {
   base(int);
};

struct sub : base {
   sub(int);
   int var;
};

sub::sub(int a) : base(a),, var(a) { }
                        // ERROR: Extra comma
```

binary exponent part required for hex floating constants

(C/C++) The exponent is missing from a hexadecimal floating-point constant. A hexadecimal floating-point constant comprises an optional sign, the `0x` prefix, a hexadecimal significand, the letter `p` to indicate the start of the exponent, a binary exponent, and an optional type specifier. These are valid hexadecimal floating-point constants:

```
0x1.FFFFFEp127f
0x1p-23
-0x1.2ACp+10
```

'bit' is not a legal align specifier

(Symantec Rez) You can't align a resource on a bit boundary. Only `nibble`, `byte`, `word`, and `long` are valid `align` specifiers.

bitstring is a numeric type

(Symantec Rez) Even though it sounds like a string, it is actually a numeric type. A bitstring is used to define a sequence of bits. The type `bitstring[16]` is the same as an integer.

blank arguments are illegal

(C/C++) Arguments are missing from a macro reference that is defined to take them. For example:

```
#define TWICE(x) (x+x)

TWICE(10)    // OK
TWICE()      // ERROR
```

'break' is valid only in a loop or switch

(C/C++) The `break` statement can occur only within a `for`, `while`, `switch`, or `do/while` statement.

can only delete pointers

(C++) The `delete` operator works only on pointers. Use `delete` on a pointer to an object and not the object itself.

can't assign to const variable

(C/C++) A new value is assigned to a `const` variable. Remove the assignment or remove the restriction from the variable.

can't calculate size of an expression that contains an array statement

(Symantec Rez) Statements that contain references to forward labels can't contain array statements.

can't calculate size of an expression that contains a switch statement

(Symantec Rez) Statements that contain references to forward labels can't contain `switch` statements.

can't declare member of another class *'identifier'*

(C++) In a class declaration, a class name modifies a member function name. For example:

```
class X {
   void func_in_X();
};
class Y {
   void X::func_not_in_X();    // ERROR
   int func_in_Y();            // OK
};
```

can't handle constructor in this context

(C++) Having a constructor as a default function parameter is illegal. For example, instead of:

```
class X {
public:
   X(int);
};
   foo(X x=X(1));          //ERROR
```

use:

```
void foo()
{
   X x(1);
   .
   .
   .
}
void foo(X x)
```

can't have data statement without matching type statement

(Symantec Rez) If you have a statement in a resource block, then there must be at least one `type` statement in the matching type block.

can't have unnamed bit-fields in unions

(C/C++) Using an unnamed bit-field in a union is illegal. Use a named bit-field or remove the bit-field.

can't nest comments

(C/C++) Warning. Avoid nesting comments; it's easy to nest incorrectly and accidentally comment out the wrong code. Instead, use `#if 0` and `#endif` to block out sections of code. Avoid crossing existing `#if`. For example, the following statements comment out the enclosed code:

```
#if
.
.
.
.
#endif
```

can't pass const/volatile object to non-const/volatile member function

(C++) An object declared as `const` or `volatile` is trying to call a member function that is not. Declare the member function `const` or `volatile`, or remove the restriction from the object. For example:

```
struct A {
   int regular_func();
   int const_func() const;
};

void main()
{
   const A const_obj;
   A regular_obj;

   const_obj.regular_func();      // ERROR
   const_obj.const_func();        // OK
   regular_obj.const_func();      // OK
   regular_obj.regular_func();    // OK
}
```

can't return arrays, functions or abstract classes

(C++) A function cannot return an array, function, or abstract class. However, a function can return a pointer to an array, a pointer to a function, or a pointer to an abstract class. For example:

```
typedef char ARRAY[256];
ARRAY func_returning_array();     // ERROR
ARRAY *func_returning_ptr_to_array();
                                  // OK
class X*func_returning_abstract_class();
                                  // OK
```

can't take address of register, bit-field, constant or string

(C/C++) You cannot take the address of a register variable, a bit-field in a structure, a constant, or a string. Declare the object differently, or avoid taking its address.

can't take sizeof bit-field

(C/C++) It is illegal to use `sizeof` to determine the size of a bit-field member of a `struct`.

cannot convert *'identifier*'* to a private base class *'identifier*'*

(C++) A pointer to a `class X` cannot convert to a pointer to a private base `class Y` unless the current function is a member or a friend of `X`.

```
class Y {
};
class X: Y;
void f(void)
{
   class X*Px;
   class Y*Py;
Py=(class Y *)Px;
}
```

cannot create instance of abstract class *'identifier'*

(C++) An abstract class contains at least one pure virtual function by the declaration `virtual func() = 0`. It is illegal to declare objects of such a class. For example:

```
class abstract_class {
public:
   virtual int func() = 0;
   int x, y;
};

class subclass : abstract_class {
public:
   virtual int func() { return (x*2); }
   int a, b;
};

void main()
{
   subclass a;           // OK
   abstract_class b;     // ERROR
   // . . .
}
```

cannot define parameter as extern

(C/C++) `extern` is an illegal storage class for a function parameter.

cannot delete pointer to const

See ARM 8.5.3 for more information.

(C++) Using the delete operator on a `const` pointer is illegal. Remove the `const` by casting, or remove the delete operator.

cannot find constructor for class matching *name*

(C++) The compiler cannot find a constructor that matches the current initializers. Use different initializers. Coerce some initializers so they match those of a constructor, or define a new constructor. For example:

```
struct X {
   X(char *);
};

void main()
{
   X a = 1L;          // ERROR
   X b = 3.1e20;      // ERROR
   X c = "hello";     // OK
}
```

cannot generate *'identifier'* for class *'identifier'*

See ARM 12.1 and 12.8 for more information.

(C++) The compiler cannot define a copy constructor `X::X(X&)` for `class X` or an assignment operator `X& operator=(X&)` for `class X` for the class. If a class needs these methods, define them explicitly.

The compiler cannot define an assignment operator if one of these conditions is true:

- The class has a `const` member or base.
- The class has a reference member.
- The class has a member that is an object of a class with a private *operator=()*.
- The class is derived from a class with a private *operator=()*.

The compiler cannot generate a copy constructor if one of these conditions is true:

- The class has a member that is an object of a class with a private copy constructor.
- The class is derived from a class with a private copy constructor.

cannot generate template instance from #pragma template identifier
(C++) The compiler cannot generate a template instance from the specifier in the `#pragma` template directive. Include the template definition in the program and spell the template instance correctly.

cannot have member initializer for *'identifier'*
(C++) The constructor initializer can initialize only non-static members.

cannot implicitly convert
(C/C++) This expression requires the compiler to perform an illegal implicit type conversion. To perform this conversion, explicitly cast the expression.

cannot raise or lower access to base member *'identifier'*
See ARM 11.3 for more information.
(C++) Access declarations in a derived class cannot grant or restrict access to an otherwise accessible member of a base class. For example:

```
class base {
public:
   int a;
private:
   int b;
protected:
   int c;
};

class sub : private base {
public:
   base::a;     // OK
   base::b;     // ERROR: can't make b
                //        accessible
protected:
   base::c;     // OK
   base::a;     // ERROR: can't make a
};              //        inaccessible
```

case *number* was already used
(C/C++) This value already occurs as a case within the `switch` statement.

casting from incomplete structure type identifier
(C++) This warning is received when the struct type being cast from is an incomplete type. This will cause a problem when it is later or

elsewhere defined as a derived class of the type being cast to. For example:

```
struct X;
struct Y;
Y * f(X *x)
{
   return ((Y*)x);
}
```

casts and sizeof are illegal in preprocessor expressions
(C/C++) A Symantec extension to ANSI C allows the use of the `sizeof` operator and performs a cast in preprocessor directives. Turning on the Strict ANSI conformance option on the Language Settings page disallows use of these expressions in a preprocessor directive.

char type only allows one character per string
(Symantec Rez) If the `type` statement is of type `char`, then the matching resource description statement needs to be in double quotes, but it must be zero or one byte long.

```
""     //valid char,
       //  a zero byte will be output
"a"    //valid char
"ab"   //invalid char
```

class name *'identifier'* expected after ~
(C++) A destructor is declared incorrectly. The proper name is *class::~class()*. If the class is named `X`, its destructor is `X::~X()`.

code segment too large
(C/C++) The code contribution of one file exceeds 32K.

comma not allowed in constant expression
(C/C++) It is illegal to use a comma in a constant expression or to separate numbers by commas or spaces.

const or reference *'identifier'* needs initializer
(C++) Non-extern `const` declarations or references must be initialized.

constant expression does not fit in type
(C/C++) Each constant expression evaluates to a constant in the range of representable values for its type.

constant initializer expected

(C/C++) When you are initializing a variable being declared, any nonpointer-type initializer must be either a constant or the address of a previously declared `static` or `extern` item. For example:

```
const float pi = 3.1415;
float a = 3.0;
static float b;

float w = a*2;     // ERROR: a isn't const
float x = pi*pi;   // OK: pi declared const
float *z = &b;     // OK: b is static
```

'continue' is valid only in a loop

(C/C++) A `continue` statement occurs out of context. Use it only within `for`, `while`, and `do/while` statements.

cstring too long (will be truncated)

(Symantec Rez) Warning. If no length indicator is given, a `cstring` appends a null byte to the string. The maximum size of a `cstring` is 2,147,483,647, but Symantec Rez will run out of memory long before then.

Data member '*identifier*' cannot appear in a struct after a dimensionless array

(C/C++) A dimensionless array can only be the last member in a structure. For example:

```
struct A {
   int X[];
   int i;          //ERROR
}
```

data or code *'identifier'* defined in precompiled header

(C/C++) Precompiled headers can contain only declarations, not definitions.

data statement type does not match type statement declaration

(Symantec Rez) Each data statement must have a type statement of the same base type.

declarator for 0 sized bit-field

(C/C++) A bit-field must have a size.

'default:' is already used
(C/C++) The `default:` statement appears more than once in a `switch` statement.

delete[] *'identifier'* not allowed for handle/Pascal class
(C++) You cannot use delete on an array of Pascal or handle-based objects.

different configuration for precompiled header
(C/C++) The precompiled header being used is precompiled with different options. Precompile the header again with the current options or check the current options for accuracy. Can also occur when a newer compiler is used with an older precompiled header.

divide by 0
(C/C++) A constant expression tries to divide by zero or get modulo (`%`) of zero.

duplicate direct base class '*identifier*'
(C++) A class cannot derive the same base class more than once (directly). For example:

```
class B {
   int i;
};
class C:B,B           //ERROR
{
};
```

empty declaration
(C/C++) A declaration must declare at least a declarator, a tag, or the members of an enumeration.

end of file found before '#endif'
(C/C++) Missing `#endif` causes the compiler to reach the end of the file in the middle of a conditional compilation statement list.

end of file found before end of comment, line *number*
(C/C++) A missing `*/` causes the compiler to reach the end of the file in the middle of a comment.

end of line expected

(C/C++) Using the Strict ANSI conformance option on the Language Settings page does not allow any text to follow the `#endif` keyword, unless the text is a comment. For example:

```
#ifdef DEBUG
   printf ("oops\n");
#endif DEBUG          // Not ANSI-compatible

#ifdef DEBUG
   printf ("oops\n");
#endif   // DEBUG     // ANSI-compatible
```

error opening resource file for $$resource or $$read statement

(Symantec Rez) The `$$resource` and `$$read` statements require Symantec Rez to be able to open the resource fork of a file. Make sure that the file exists, that it is not opened with write access by another running application, and that it is in the system or project tree.

escape sequence doesn't fit in a byte

(Symantec Rez) An escape sequence is a series of two or more characters that form a character. Octal, hexadecimal, decimal and binary escape sequences can be used to specify characters that do not have predefined escape sequences. However, all escape sequences must fit into a single byte. Escapes must be of the form:

base	number form	digits	example
2	\0Bbbbbbbbb	8	\0B10000001
8	\000	3	\101
10	\0Dddd	3	\0D065
16	\0Xhh	2	\0X41
16	\$hh	2	\$41

Here are some examples:

```
\077            /* 3 octal digits*/
\0xFF           /* '0x' plus 2 hex digits*/
\$F1\$F2\$F3    /* '$' plus 2 hex digits*/
\0d099          /* '0d' plus 3 decimal digits*/
```

Remember that an octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write `AB\007CD`, not `AB\7CD`.

Also, remember that Symantec Rez is not C or C++. The string "\0" is not valid in Symantec Rez because the \0 character is not a valid escape sequence.

expected data definition of *'identifier'*, not func definition
(C/C++) Declarations appearing outside a function body must have correct declaration syntax. This error occurs most often when too many braces ({ }) have been specified in a function body.

expected point type data statement
(Symantec Rez) As Symantec Rez parses your resources, it knows when to expect certain kinds of statements. You will get this error if Symantec Rez expects to parse a point statement, and sees something other than a point statement.

expected switch statement
(Symantec Rez) As Symantec Rez parses your resources, it knows when to expect certain kinds of statements. Make sure you picked a valid case from the switch statement in the matching type block.

exponent expected
(C/C++) The compiler cannot find the exponent for the floating-point number written. Do not put any white space between the e and the following exponent.

expression expected
(C/C++) The compiler expects to find an expression but cannot. A missing semicolon or close brace may cause this problem.

external with block scope cannot have initializer
(C/C++) Initializing a variable declared extern is illegal. Instead, initialize the variable in the file where it is defined.

field *'identifier'* must be of integral type
(C/C++) An inappropriate type occurs for a bit-field member of a structure. Use signed/unsigned char, short, int, or long.

file specification string expected
(C/C++) The compiler cannot find the filename string in an #include statement. Enclose the filename in double quotes or angle brackets.

forward referenced class *'identifier'* cannot be a base class

(C++) A class must be declared before it can be used as a base class for a new class. A forward declaration is not sufficient. For example:

```
class A;        // Forward reference for A
class B {       // Declaration of B
   int a, b, c;
   void f();
};

class X : A { /*...*/ };// ERROR: A isn't
                        //       declared

class Y : B { /*...*/ };// OK: B is
                        //    declared
```

Function definitions with separate parameter lists are not allowed in C++

(C++) In C++, function definitions with separate parameter lists are illegal. If ANSI is off, this error becomes a warning. For example:

```
void f(a)
int a;
{
   .
   .
}
```

Note: This also happens when `typedef` names are missing or misspelled. For example:

```
typedef int PARAM_TYPE;
void f(PARM_TYPE a)
{
   .
   .
}
```

Function definitions with separate parameter lists are obsolete in C++

(C++) Warning. In C++, function definitions with separate parameter lists are illegal. See previous error message for examples.

function *'identifier'* has no prototype

(C/C++) The compiler cannot find a function prototype for this function. The C++ compiler requires function prototypes by default.

function *'identifier'* is too complicated to inline
(C/C++) Warning. A function declared as `inline` is too complex to compile inline, so the compiler compiles it as a normal function.

function definition must have explicit parameter list
See ARM 14.1 for more information.
(C/C++) A function definition requires an explicit parameter list. It cannot inherit a parameter list from a `typedef`. For example, this definition does not compile:

```
typedef int functype(int q, int r);

functype funky // ERROR: No explicit
{              //        parameter list
   return q+r;
}
```

function expected
(C/C++) The compiler expects to find a function declaration but does not. Check for mismatched braces, parentheses not preceded by a function name, or a template declaration not followed by a class or function declaration.

function member *'identifier'* cannot be in an anonymous union
(C++) Anonymous unions cannot have function members.

functions can't return arrays or functions
(C) It is illegal for a function to return an array or a function. See "can't return arrays, functions or abstract classes."

global anonymous unions must be static
See ARM 9.5 for more information.
(C++) Anonymous unions must be `extern` or `static`.

hex digit expected
(C/C++) The compiler expects to find a hexadecimal digit after the characters `0x`. Do not put any white space after the `x`.

hex strings must be an even number of digits
(Symantec Rez) All hex strings must be an even number of digits. Symantec Rez will not pad the string for you.

identifier expected
(C/C++) The compiler expects to find an identifier, but finds instead another token.

identifier found in abstract declarator

(C/C++) A type in a `sizeof` expression, `typedef` statement, or similar place incorrectly includes a variable name. For example:

```
x = sizeof(int a[3]);
                    // ERROR: a is a variable
                    //        name.
x = sizeof(int[3]);
                    // OK
```

identifier is longer than 1024 chars

(C/C++) The maximum size of an identifier is 1024 characters.

identifier or '(declarator)' expected

(C/C++) The compiler expects to find a declaration for a static variable, an external variable, or a function. If this error appears in a function, see if there are more right braces than left braces.

illegal bitstring size

(Symantec Rez) The only numeric type specification that allows a size is `bitstring`. The `byte`, `integer`, and `longint` types cannot have sizes.There is no implicit array type as in C. If you want an array of bytes, you have to explicitly declare them:

```
type 'test' {
   array [12] {
      byte; //12 bytes
   };
};
```

illegal cast

(C/C++) You cannot cast structs or unions to other types. You can cast numerical values or pointers to other numerical values or pointers. Example:

```
typedef struct {short v, h;} Point;
void function(void)
{
  Point p;
  long l;
  p = (Point) l;          // NO
  p = *(Point *) &l;      // OK
}
```

illegal character, ascii *number* decimal

(C/C++) The source file includes a character outside a comment or string, such as `@` or `$`, that is not part of the C character set.

illegal combination of types

(C/C++) Certain types cannot occur together. For example, you cannot declare a variable to be a `short long int`.

illegal constructor or destructor declaration

(C++) A constructor or destructor is declared incorrectly. For example, a constructor may be declared as `virtual` or `friend`, a destructor may be declared as `friend`, or a return value may be specified for a constructor or destructor.

illegal hex string character

(Symantec Rez) Escape sequences are not allowed inside hex strings.

illegal operand types

(C/C++) The operands are of the wrong type. Cast the operands to the correct type.

illegal pointer arithmetic

(C/C++) You cannot perform some arithmetic operations on pointers. You cannot assign integers to pointers except for the constant zero. In addition, you cannot compare pointers to integers, again, with the exception of the constant zero. In all cases, you can use a cast to force the operation when necessary. Example:

```
void function(void)
{
  short *short_ptr;
  short i;
  char *p1, *p2;
  long result;

  short_ptr = 0x220;      // NO
  short_ptr = 0;          // OK
  short_ptr = (short *) 0x220;
                          // OK: Address of
                          //     MemErr
  result = p1 + p2;       // NO: Can't add
                          //     pointer
  result = p1 / p2;       // NO: Can't divide
}                         //     pointers
```

illegal rect statement

(Symantec Rez) If the `type` statement called for a rectangle, then your array must contain exactly four elements.

```
{ 0, 0, 100, 100 };      /* valid rect */
{ 0, 0 };                /* illegal rect */
```

illegal redefinition of macro 'symbol'
illegal resource attribute
(Symantec Rez) Symantec Rez can't set the changed bit of a resource.

illegal return type for operator->()
See ARM 13.4.6 for more information.
(C++) `operator->()` must return one of these:

- A pointer to an object of the class that defines `operator->()`
- A pointer to an object of another class that defines `operator->()`
- A reference to an object of another class that defines `operator->()`
- An object of another class that defines `operator->()`

illegal type for *'identifier'* member
(C/C++) Variables cannot be of type `void`.

illegal type id range
(Symantec Rez) If a range is provided by a type declaration, then all resources of that type must be within that range. It is legal to have several type declarations of various ranges, but all resources must be within one of those ranges.

```
type 'test' (0:128){
   pstring;
   rect;
};
type 'test' (129:131){
   pstring;
   point;
};

resource 'test'(128) {//legal - falls within
                      // range of first
                      // 'test' type

   "hello world",
   { 0, 0, 100, 100 };
};

resource 'test' (1000) {//no type exists for
                        // 'test' id = 1000

   "hello world",
   { 0, 0 };
};
```

Illegal use of template type argument during expansion of template identifier

(C++) The template type argument must not appear in the template class declarator. For example:

```
template<class T> class X<T>{     //ERROR
};
```

inherited function must be member of derived class

(C++) When using the inherited `::`, the member being accessed must exist in the first base class of the specified object.

initialization of *'identifier'* is skipped

(C++) It is illegal in C++ to skip over an initialization. For example:

```
switch (i) {
case 1:
   int x = 3;
case 2:
   break;       //ERROR: initialization
                //       of x is skipped.
```

initializer for static member must be outside of class def

See ARM 9.4 for more information.

(C++) Static class members must initialize outside the class definition. For example:

```
class A {
   static int a = 5;
          // ERROR: Can't initialize static
          //        class var in class def.
   void f();
};

class B {
   static int b;
   void f();
};
int B::b = 6;
          // OK: Initialize static class var
          //     outside class def.
```

integer constant expression expected

(C/C++) An integer constant expression must occur in `case` statements, in array size declarations, and in the `#if`, `#elif`, `#exit`, and `#line` preprocessor commands.

integral expression expected

(C/C++) An integer type must occur in `case` statements, in array size declarations, and in the `#if`, `#elif`, `#exit`, and `#line` preprocessor commands. For example:

```
float f;
   f=f<<1;
```

internal error *'filename'* line *number*

(C/C++) This indicates a defect in the Symantec C++ compiler. Please contact Symantec technical support with details of this problem, including the filename and line number reported.

invalid escape character

(Symantec Rez) Symantec Rez does not allow just any character to be escaped. It expects them to be formatted as octal, hex, decimal or binary escapes. An escape sequence is a series of two or more characters that form a character. Octal, hexadecimal, decimal and binary escape sequences can be used to specify characters that do not have predefined escape sequences. However, all escape sequences must fit into a single byte. Escapes must be of the form:

Base	Number form	Digits	Example
2	\0Bbbbbbbbb	8	\0B10000001
8	\000	3	\101
10	\0Dddd	3	\0D065
16	\0Xhh	2	\0X41
16	\$hh	2	\$41

Here are some examples:

```
\077            /* 3 octal digits*/
\0xFF           /* '0x' plus 2 hex digits*/
\$F1\$F2\$F3    /* '$' plus 2 hex digits*/
\0d099          /* '0d' plus 3 decimal digits*/
```

Remember that an octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write `AB\007CD`, not `AB\7CD`.

Symantec Rez is not C or C++; the string `"\0"` is not valid in Symantec Rez, because the `\0` character is not a valid escape sequence.

invalid parameter to #print directive
(Symantec Rez) The `#printf` directive can be used to print strings and integral numbers. It can't be used to print array values or label indexes.

invalid reference initialization
See ARM 8.4.3 for more information.

(C++) Results from trying to initialize:

- A `volatile` reference to a `const`
- A `const` reference to a `volatile`
- A plain reference to a `const` or `volatile`

invalid storage class for friend
(C/C++) Friend functions cannot be virtual.

last line in file had no \n
(C/C++) Compiling with the Strict ANSI conformance option on means that the last line of a source file must end with a newline character. A backslash cannot precede the newline.

line number expected
(C/C++) The line number in the `#line` directive must be a constant expression.

linkage specs are "C", "C++", and "Pascal", not "*identifier*"

(C++) The compiler supports only the C++, C, and Pascal linkage types.

local class cannot have static data on non-inline function member *'identifier'*

See ARM 9.4 for more information.

(C++) A local class (that is, a class declared within a function) cannot have a static data member or a non-inline function member. For example:

```
void f()
{
   class local_class {
      int a, b;
      static int c;  // ERROR: Can't have
// static var in
                     // local class
      void g();      // ERROR: non-inline
   } l1, l2;         // function

   // . . .
}
```

lvalue expected

(C/C++) The compiler expects to assign a value to an expression, such as a variable. For example:

```
short short_f(void);
short *pshort_f(void);
void function(void)
{
  short i;
  short *p = &i;

  // Operand of ++ must be an lvalue
  7++;             // NO
  short_f()++;     // NO
  pshort_f()++;    // NO

  // Left operand of an assignment
  // must be an lvalue.
  pshort_f() = i; // NO
  *pshort_f() = i;// OK: Produces an lvalue
  (*p)++;         // OK
  (*pshort_f())++;// OK
}
```

macro *'identifier'* can't be #undef'd or #define'd

(C/C++) It is illegal to redefine or undefine this predefined macro.

main() cannot be static or inline

See ARM 3.4 for more information.

(C++) It is illegal to declare the function `main()` as `static` or `inline`.

malformed template declaration

See ARM 14 for more information.

(C++) A template class or function is declared incorrectly. The following are correct declarations:

```
template<class T, int x>    // OK
class vector {
   T v[x];
public:
   vector();
   T& operator[](int);
   /* . . . */
};

template<class T>           // OK
T ave (T x, T y) {
   return ( (T)((x+y)/2) );
}
```

Maximum number of *number* nested template expansions exceeded for expansion of '*identifier*'

(C/C++) Internal compiler limit on number of nested template expansions was exceeded.

maximum length of *number* exceeded definition

(C/C++) A macro was seen that was larger than the compiler's internal buffer.

maximum of number characters in string exceeded

(C/C++) A string literal cannot exceed 1024 characters.

maximum width of *number* bits exceeded

(C/C++) This field can contain *number* bits. For example:

```
struct X {
   char  x:9;  // ERROR: char is 8 bits
   short y:17; // ERROR: short is 16 bits
   long  z:33; // ERROR: long is 32 bits
};
```

member *'identifier'* can't be same type as struct *'identifier'*

(C/C++) A structure cannot contain itself as a member, as in:

```
struct X {
   struct X x;
};
```

member *'identifier'* is const but there is no constructor

(C++) If a class has a `const` member, the class must also have a constructor. Initialize a `const` variable only in the constructor, for example:

```
class A {          // ERROR: no constructor
   const int x;    //        to initialize x
   int y, z;
   void f();
};

class B {
   const int x;
   int y, z;
   void f();
   B();            // OK: x can be
};                 //     initialized.
```

member *'identifier'* of class *'identifier'* is not accessible

(C++) A class member that is private or protected cannot be accessed.

member *'identifier'* of class *'identifier'* is private

(C++) Only a class function or a derived function of the class can use a private member. For example:

```
class super {
private:
   int x;
   int f();
};

class sub : super {
   int g();
};

int super::f()
{
   return (x++);   // OK: B::f() is a
}                  //     member function
```

```
int sub::g()
{
   return (x++);   // ERROR: sub::g() is a
}                  //        member function
                   //        of a derived
                   //        class
main()
{
   super s;
   s.x = 3;        // ERROR: main() isn't a
   return 0;       //        member function
}                  //        or a friend
                   //        function
```

member functions cannot be static

(C++) If you use the ANSI conformance option, you cannot declare a member function to be static.

missing ',' between declaration of '*identifier*' and '*identifier*'

(C/C++) Declarations must be separated by a comma. This often occurs when a `typedef` is missing or misspelled.

missing 'key' in case selector

(Symantec Rez) Every case in a `switch` statement must have a `key` statement. It is this `key` statement that allows Symantec Rez to select the correct case. Which case applies is based on the key value, for example,

```
type 'DITL' {
   ...type specifications...
   switch {
      case Button:
         boolean        enabled, disabled;
         key bitstring[7] = 4;/* key value */
         pstring;
      case CheckBox:
         boolean        enabled, disabled;
                  /* missing key statement */
         pstring;
      case ...
   };
};
```

missing or undefined label or enumerated value

(Symantec Rez) Symantec Rez expected an identifier, but got something else like a ',' or '}'.

must be void operator delete(void * [,size_t]);

(C/C++) The improper prototype occurs when the `delete` operator for a class that uses the C++ model is overloaded. The prototype for an operator delete overload must be either:

```
void operator delete(void *);        // OK
```

or

```
void operator delete(void *,size_t);// OK
```

must be void operator delete(void**)**

(C++) You can override `new` and `delete` for Pascal classes, but the overridden functions have different arguments from those for other classes. Pointers are of type `void**`, not `void*`.

must use delete[] for arrays

See ARM 5.3.4 for more information.

(C++) To delete an array `a`, use this statement:

```
delete[] a; // OK
```

and not

```
delete a;    // ERROR
```

need at least one external definition

(C/C++) ANSI requires a translation unit to define at least one external name.

new *identifier* [], not allowed for handle/Pascal class

(C++) You cannot allocate an array of Pascal or handle-based objects using `new`.

no constructor allowed for class *'identifier'*

(C++) The class includes a variable with the same name as the class. This prevents the use of a constructor that must have that name.

no definition for static *'identifier'*

(C/C++) A static function was declared but never defined. If this error occurs for a template function, see "§14.1 Templates" on page 218.

```
static void f(void);
void  g(void)
{     f();
}
```

Note: In C++, this also occurs for template functions that are not defined when using `#pragma template_access static`. For example:

```
#pragma template_access static
template<class T> T f(T);
void g()
{
   int i;
   i = f(7);
}
            //ERROR: no definition for
            //       static 'f(int)'
```

no identifier for declarator

(C/C++) An identifier is missing from this declaration. For example:

```
void f(char [3])  // ERROR: No identifier
{
   // . . .
}

int [3];          // ERROR: No identifier
int a[3];         // OK: Identifier is a
```

no instance of class *'identifier'*

(C++) You get this error message for attempting to reference class members in a class static function.

no instance of class *'identifier'* for member *'identifier'*

(C++) It is illegal to attempt the following:

- Call a nonstatic member function without using an instance of the class
- Access a nonstatic data member without using an instance of the class
- Define a nonstatic data member outside a class

However, it is legal to attempt the following:

- Call a `static` member function without an object
- Access a `static` data member without an object
- Define a `static` data member outside a class

For example:

```
struct CLASS {
   static void static_func();
   void nonstatic_func();

   static int static_data;
   int nonstatic_data;
};

int CLASS::nonstatic_data = 1;    // ERROR
int CLASS::static_data = 1;       // OK

void main()
{
   CLASS object;

   int i = CLASS::nonstatic_data;// ERROR
   int j = object.nonstatic_data;// OK

   CLASS::nonstatic_func();       // ERROR
   CLASS::static_func();          // OK
   object.nonstatic_func();       // OK
}
```

no match for function *'identifier'*

(C++) The function is overloaded and the compiler cannot find a function that matches the call.

no resources read by include statement

(Symantec Rez) Warning. If an `include` statement fails for any other reason other than running out of memory, you will get this error message. Make sure that the file exists, that it is not opened with write permission by any other application, and that you have specified a valid pathname.

no resources read by read statement

(Symantec Rez) Warning. If a `read` statement fails for any other reason other than running out of memory, you will get this error message. Make sure that the file exists, that it is not opened with write permission by any other application, and that you have specified a valid pathname.

no return value for function *'identifier'*

(C++) A function has a return type other than `void`, but it has no return statement or has a path by which it doesn't return. For example:

```
int f()
{
   if (x)
      return (1);
}
```

no such option *'symbol'*

(C/C++) You used the identifier *symbol* as an argument to `#pragma options` or `__option`, but it is not one of the valid options. Example:

```
#pragma options(!optimize)          // NO
#pragma options(!global_optimizer)  // OK
```

no tag name for struct or enum

(C/C++) Warning. If a `struct` or an `enum` does not have a tag name, further objects of this type cannot be declared later in the program. Give every `struct` and `enum` a tag name so the compiler's type-safe linkage system can use it.

non-const reference initialized to temporary

See ARM 8.4.3 for more information.

(C++) Warning. In most cases, this message means that a reference is being bound to a temporary due to type conversion. Since the reference is not `const`, the referenced temporary may change its value.

However, this message becomes an error when the Strict ANSI conformance option on the Language Settings page is set.

Non-local name *'identifier'* cannot be declared in a template instantiation.

(C++) A template instantiation cannot introduce a new name into the global scope. For example:

```
template<class T> class X {
   friend int operator == (T t1, T t2)//ERROR
   {
      return ((long)t1 == (long)t2);
   }
};
X<int> X;
```

Use the following instead:

```
template<class T> int operator == (T t1, T t2)
{
   return ((long)t1 == (long)t2);
}
template<class T> class X {
   friend operator == (T t1, T t2);
};
X<int> X;
```

For more information, see "§14.7 Friends" on page 224.

not a struct or union type

(C/C++) The type of object preceding the object member operator selector (`.`) or the pointer to object selection (`operator ->`) is not a `class`, a `struct`, or a `union`.

not an overloadable operator token

(C++) You cannot overload these operators:

```
.          .*         ::         ?:         sizeof
#          ##
```

not enough bits to use $$byte, $$word or $$long

(Symantec Rez) If the resource is smaller than 8 bits, you can't use `$$byte`. Similarly, you need 16 bits and 32 bits to use `$$word` and `$$long`.

not in a switch statement

(C/C++) It is illegal to use a `case` or `default` statement outside a `switch` statement.

number *'number'* is too large

(C/C++) The number is too large to be represented in an object with `long` type.

number is not representable
(C/C++) The compiler cannot represent a numeric constant because of the constraints listed in the following table:

You cannot represent...	If it is...
Integer	Greater than `ULONG_MAX` (in `limits.h`)
Floating-point number	Less than `DBL_MIN` or greater than `DBL_MAX` (in `float.h`)
Enumeration constant	Greater than `INT_MAX` (in `limits.h`)
Octal character constant	Greater than `255`

Table B-1 Unrepresentable numbers

object has 0 size
(C/C++) You cannot subtract pointers to objects of 0 size.

octal digit expected
(C/C++) The compiler expects that a number with a leading `0` is an octal digit. Using an `8` or a `9` is illegal.

one argument req'd for member initializer for *'identifier'*
(C++) Member initializers in which the member lacks a constructor must have exactly one parameter because the member is initialized by assignment.

only one identifier is allowed to appear in a declaration appearing in a conditional expression
(C++) When declaring identifiers in `if`, `for`, `while`, and `switch` statements, only one identifier is allowed.

only classes and functions can be friends
(C++) It is legal to declare other classes or functions `friend` only when declaring a function within a class.

only pointers to handle based type allowed

(C++) You cannot declare an instance of a handle object. For example:

```
class __handle X
{
}
x y;
```

operator functions ->() and [] must be non-static members

(C++) It is illegal to declare as static these operators:

- The pointer to object selection operator (->)
- The function call operator (())
- The array operator ([])

operator overload must be a function

(C++) It is illegal to declare an overloadable operator as a variable. For example:

```
struct X {
   int operator<<;    // ERROR
};
```

out of memory

(C/C++) The compiler is out of memory. Try the following:

- Break the file or function into smaller units
- Increase the partition size for the Symantec Project Manager
- Close any open windows in the editor

overloaded function *'identifier'* has different access levels

See ARM 11.3 for more information.

(C++) It is illegal to adjust the access of an overloaded function that has different access levels. For example:

```
class base {
public:
   void f(int);
private:
   void f(float);
};

class sub : base {
   base::f;           // ERROR: f() is
};                    //        overloaded.
```

overloading type conversion or operator function not allowed
(C++) Pascal object classes do not allow overloaded functions or operators.

parameter list is out of context
(C/C++) Parameters in a function definition are illegal and are discarded. For example:

```
int f(a, b);          // ERROR
int g(int, int);      // OK
int h(int a, int b); // OK
```

parameter lists do not match for template *'identifier'*
(C++) The parameter list for the template instantiation does not match the formal parameter list for the class definition.

```
template<class T, int size> class vector;
template<class T, unsigned size> class
 vector;              // no {}
vector<int,20> x; // OK
vector<float,3.0> // ERROR: 3.0 is not an
                  //        int.
```

Pascal object class expected
(C++) You cannot use C++ virtual functions in a code resource.

pointer required before '->' or after '*'
(C) These operators can apply only to pointers. The operators `->` and `*` must be used with a pointer.

pointer required before '->', '->*' or after '*'
(C++) These operators can apply only to pointers. The operators `->`, `->*` and the operator `*` must be used with a pointer.

pointer to member expected to right of .* or ->*
(C++) The identifier after `.` or `->*` must be a pointer to a member of a `class` or `struct`.

pointers and references to references are illegal
(C++) You cannot declare a pointer or reference to reference type, as in:

```
int & & a;
```

possible extraneous ';'

(C/C++) Warning. The compiler finds a semicolon immediately after an `if`, `switch`, or `while` statement and executes the next statement, regardless of whether the test evaluates to true or false. For example:

```
int x=1, y=0;

if (x==y);              // WARNING: Extra
   printf("x==y\n");    // semicolon. printf()
                        // always executed.

if (x==y)               // OK
   printf("x==y\n");
```

If you want a semicolon, suppress the warning by putting white space, such as a space or a return, between the close parenthesis and the semicolon.

```
while (fread(file)==unwanted_data)
   ;                    // OK: semicolon is
                        //     intentional
```

possible unintended assignment

(C/C++) Warning. The assignment operator (=) instead of the equality operator (==) appears in the test condition of an `if` or a `while` statement. For example:

```
if (x=y) {. . .}   // WARNING: x=y  is an
                   //          assignment
```

instead of

```
if (x==y) {. . .} // OK: x==y is a test
```

Test the value of the assignment explicitly, like this:

```
if ((x=y) != 0) {. . .}
                  // OK: (x=y)!=0 is a test
```

The compiler produces identical code for the first and third examples.

premature end of source file

(C/C++) A string that is missing a close quote or a comment that is missing a `*/` causes the compiler to reach the end of the file while processing a comment.

prior forward reference class *identifier* must match handle/Pascal class type

(C++) This error occurs when there is a mismatch between a forward declaration of a class and a definition, as in:

```
class x;
class __pasobj x {
    .
    .
    .
}
```

prototype for *'identifier'* should be *identifier*

(C/C++) A function of the form: `func(s) short s; { ... }` should be prototyped as:

```
func(int s);
```

rather than:

```
func(short s);
```

pure function must be virtual

(C++) Pure member functions must be declared as `virtual`, like this:

```
class B {
   virtual void f() = 0;        // OK
   void g() = 0;                // ERROR
};
```

qualifier or type in access declaration

See ARM 11.3 for more information.

(C++) It is illegal to specify a storage class or type when adjusting the access to a member of a base class. For example:

```
class base {
public:
   int b, c, d;
   int bf();
};

class sub : private base {
   int e;
public:
   base::b;           // OK
   int base::c;       // ERROR
   static base::d;    // ERROR
};
```

redefinition of default parameter

(C++) It is illegal to redefine the default argument for a parameter even if it is redefined to the same value. For example:

```
// Prototyping the function.
int f(int, int=0);

// Defining the function.
int f(int a, int b=0)// ERROR: Can't
{                    // redefine default
   return g(a,b);    // argument, even to
}                    // the same value.
```

The line given for the error is sometimes past the close brace of the body of the function.

resource type has already been declared

(Symantec Rez) Symantec Rez allows you to decide if you want to allow redeclaration of resource types. This option can be set in the **Symantec Rez options** dialog. The error message is generated only when this option is turned off.

resource types have to be exactly four characters long

(Symantec Rez) All resource types must be exactly four characters long. The most common way to declare a resource type is to use single quotes around four characters.

```
'test'   //valid resource type
'tests'  //invalid resource type
```

return type cannot be specified for conversion function

See ARM 12.3.2 for more information.

(C++) It is illegal to specify the return type of a conversion function. For example:

```
class X {
   char* operator char* ();    // ERROR
   operator char* ();          // OK
};
```

returning address of automatic *'identifier'*

(C/C++) This results in an invalid pointer beyond the end of the stack. When the function returns, the caller receives an illegal address that can cause a bus error.

should be *number* parameter(s) for operator

(C++) The incorrect number of arguments appears in a declaration of an overloaded operator. The function call operator `()` is *n*-ary; it can take any number of arguments.

size of *identifier* is not known

(C/C++) It is illegal to use a `struct` or an array with an undefined size. For example:

```
struct x {
   int a[];          // ERROR
   /* . . . */
};

struct y {
   int a[100];       // OK
   /* . . . */
};
```

statement expected

(C/C++) The compiler expects a statement but does not encounter one. A missing comma or semicolon or a label without a statement can cause this error. For example:

```
while (TRUE) {
   // . . .
   if (done) goto end1;
   // . . .
end1:
}        // ERROR: No statement after
         //        label.

while (TRUE) {
   // . . .
   if (done) goto end2;
   // . . .
end2:
   ;     // OK: Null statement after label.
}
```

static function *'identifier'* cannot be virtual

(C++) Static member functions of classes cannot be virtual.

static or non-member functions can't be const or volatile

(C++) It is illegal to declare a static `class` member function or a nonmember class function as `const` or `volatile`.

static variables in inline functions not allowed
(C++) It is illegal to declare a static variable within an inline function.

storage class for *'identifier'* can't be both extern and inline
(C++) It is illegal to use the inline type specifier for a function declared external.

string expected
(C/C++) The compiler expects to encounter a string but cannot find one. Check for an `#ident` directive not followed by a string.

string too long (will be truncated)
(Symantec Rez) Warning. If no length indicator is given, a `pstring`, `wstring`, or `cstring` stores the number of characters in the corresponding data definition. However, the maximum size of a `pstring` is 255, and the maximum size of a `wstring` and `cstring` is 2,147,483,647.

struct-declaration-list can't be empty
(C) In C, a `struct` must contain at least one member. For example:

```
struct X {};
```

template-argument *'identifier'* must be a type-argument
See ARM 14.4 for more information.
(C++) In a function template, template arguments must be type arguments. Unlike class templates, function templates cannot have expression arguments. For example:

```
template<class T, int x> foo(T y)
// ERROR: x is an expression argument.
{
   return x+y;
}
```

template-argument *'identifier'* not used in function parameter types
See ARM 14.4 for more information.
(C++) When you define a function template, every template argument in the template's argument list must appear in the function's argument list. For example:

```
template<class T1, class T2>
int bar(T1 x)      // ERROR: T2 isn't in
{                  //        function's
   T2 y;           //        argument list.
   // ...
}
```

the change statement is not supported
(Symantec Rez) Warning. Because Symantec Rez is a translator in an integrated environment, the `change` statement is not supported. Symantec Rez can only compile resource files and add resource contributions to the resource fork of a project, it cannot delete, change or otherwise modify the resource fork of the project except to add to it.

the delete statement is not supported
(Symantec Rez) Warning. Because Symantec Rez is a translator in an integrated environment, the `delete` statement is not supported. Symantec Rez can only compile resource files and add resource contributions to the resource fork of a project, it cannot delete, change or otherwise modify the resource fork of the project except to add to it.

the resource type 'type' has been redeclared
(Symantec Rez) Warning. Symantec Rez allows you to decide if you want to allow redeclaration of resource types. This option can be set in the **Symantec Rez options** dialog. You will get this warning when the option is enabled, otherwise Symantec Rez will generate a fatal error and halt compilation of the file.

too many initializers
(C/C++) The number of initialization values exceeds the expected number of data items specified in the declaration of the data structure. Example:

```
char *directions[4] = {"north", "east",
      "south", "west", "lost" };     // NO
struct { short a,b,c; } x[2] =
      { 1, 2, {3, 4 } };             // NO
```

too many parameters to $$Format statement
(Symantec Rez) The `$$Format` function has a maximum of 20 parameters.

trailing parameters must have initializers
(C++) Parameters with default initializers must occur at the end of a parameter list. For example:

```
int f(int, int=1, int=0);  // OK
int g(int=0, int=1, int);  // ERROR
int h(int=0, int, int=1);  // ERROR
```

type conversions must be members
(C++) It is illegal to declare a type conversion function outside a class. Declare it inside a class.

type is too complex
(C++) The compiler appends information regarding parameter and return types to the end of a function name. With this information added, the identifier exceeds the compiler's maximum of 1024 characters.

type mismatch
(C/C++) This error is either a syntax error or a warning message. The compiler expects to find one data type but finds another. More information about which types it expects and what it finds follows this message.

type must be void **operator new(Pascal void (*) ( ), size_t)
(C++) You can override `new` and `delete` for Pascal classes, but the overridden functions have different arguments from those for other classes. Pointers are of type `void**`, not `void*`, and `new` has an additional (leading) parameter of type `pascal void (*) ()`.

type must be void *operator new(size_t [,..]);
(C++) The wrong prototype appears when the `new` operator for a class that uses the C++ model is overloaded. When operator `new` is overloaded, it must have a return type of `void *` and take a first argument of `size_t`. The compiler automatically sets the value of the first argument to be the class size in bytes.

type of *'identifier'* does not match function prototype
(C/C++) The arguments of the function do not match the prototype previously given.

type or storage class is required in the declaration of *'identifier'*
(C/C++) A declaration must include a type or storage class. This error is most often caused by a `typedef` that is either not declared or misspelled. For example:

```
typedef short OSErr;
OSerr oserr;              //ERROR
```

undefined escape sequence
(C/C++) The compiler recognizes only the following escape sequences in a string or character constant:

This sequence...	Represents...
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\\`	Backslash
`\a`	Alert (bell)
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Tab
`\v`	Vertical tab
`\x`*XXX*	The character specified with the hexadecimal number
`\000`	The character with the octal number

Table B-2 Defined escape sequences

undefined identifier *'identifier'*
(C/C++) It is illegal to use an identifier without declaring it. Spell the identifier correctly.

undefined label *'identifier'*
(C/C++) A label must be defined for the `goto` command to go to. Spell the label correctly and make sure the label appears in the same function as the `goto`.

undefined tag *'identifier'*
(C/C++) The structure or union is not defined.

undefined use of struct or union
(C/C++) It is illegal to use operators, such as arithmetic or comparison operators, on a `struct`, `class`, or `union`.

unexpected end of file
(Symantec Rez) End-of-file was reached before a Symantec Rez construct was completed. Symantec Rez expected more resource or type statements. A common error is a missing `}`. Example:

```
type 'test'{
/* EOF - end of file before close brace */
```

unexpected end of resource
(Symantec Rez) Symantec Rez expected to see at least one more statement before the closing '}'

unexpected end of data statement
(Symantec Rez) Symantec Rez expected more resource statements to match the type statements, or you might be missing a }.

union members cannot have ctors or dtors
(C++) A union cannot contain a member that is an object of a class with a constructor or a destructor.

unknown enumerated value or label
(Symantec Rez) Symantec Rez encountered an identifier that has not been defined. Make sure that you are not missing a ',' or ';'.

unknown resource type 'symbol'
(Symantec Rez) All resource types must be defined before being referenced. Make sure that the resource type falls within the range for the given type definition, and that you have included the correct header file if it is a predefined resource type.

unknown string prefix 'symbol'
(Symantec Rez) The numeric types (`bitstring`, `byte`, `integer` and `longint`) are fully specified like this:

```
[unsigned] [radix] numeric-type
      [= expr | symbol definition ...];
```

Where radix is one of the following string constants:

```
hex   decimal  octal    binary
```

If any string constant other than the word unsigned or one of the radix given above appears before the numeric type, it is considered an unknown string prefix.

unknown token
(Symantec Rez) Certain characters that are part of the ASCII character set are illegal tokens in Symantec Rez. However, any character can occur within comments, string literals or character literals. Example:

```
integer = #define M 4;      /* Illegal */
```

unrecognized pragma
(C/C++) This error occurs when a `#pragma` is seen that the compiler does not recognize. It is a warning when using `#pragma xxx` and an error when using `#pragma SC xxx`.

unrecognized preprocessor directive *'#identifier'*
(C/C++) The compiler does not support the specified preprocessor directive.

unrecognized token
(C/C++) The compiler does not recognize the token as valid. Check for an extra `U` or `L` suffix in an integer constant. It is illegal to use `$` and `@` in identifiers.

unterminated macro argument
(C/C++) A macro argument is missing a close quote or parenthesis.

unterminated string
(C/C++) A string is missing a close quote, or a file contains a lone quote mark.

use delete[] rather than delete[expr], expr ignored
(C++) Warning. This syntax for deleting an array of objects is outdated, although the current version of the compiler supports it and ignores *expr*.

```
delete [expr] p;        // WARNING: obsolete
```

New code uses this syntax instead:

```
delete [] p;            // OK
```

using operator++() (or --) instead of missing operator++(int)
(C++) Warning. It is illegal to use the postfix increment (or decrement) operator on an object of a class, such as `x++`, without overloading the postfix operator for that class. However, the prefix operator is overloaded. The compiler uses the prefix version of the operator.

To overload the postfix increment operator `x++`, use `operator++()`. To overload the prefix increment operator `++x`, use `operator++(int)`.

value of expression is not used

(C/C++) Warning. It is illegal to compute an expression without using its value, such as the equality operator (==) instead of the assignment operator (=). For example:

```
x==y;     // WARNING: The value of x
          //          doesn't change.
x=y;      // OK: x and y have same value.
```

Failure to assign the result of a computation to a variable can also cause this error. For example:

```
t-5;      // WARNING: Result of this
          //          computation is lost.
x=t-5;    // OK: x contains the result.
t-=5;     // OK: t contains the result.
```

variable *'identifier'* used before set

(C/C++) Warning. The optimizer discovers that a specified variable appears before it is initialized. The program may generate inexplicable results.

vectors cannot have initializers

(C++) It is illegal to initialize a vector of objects with a constructor that has an argument list.

very large automatic

(C/C++) Warning. Large automatic variables can cause stack overflow. Dynamically allocate the memory with a function such as `malloc()`.

voids have no value

(C) Functions declared as `void` cannot return a value.

voids have no value, ctors and dtors have no return value

(C++) It is illegal to return a value from a constructor, destructor, or function declared `void` or a reference to a `void`. It is also illegal to use the value of a constructor, destructor, or function declared `void`.

'while' expected

(C/C++) The keyword `while` is missing from the end of a `do/while` loop. For example:

```
do {
   x = f(y);
} (x!=0);          // ERROR: missing while.

do {
   x = f(y);
} while (x!=0);    // OK
```

wstring too long (will be truncated)
(Symantec Rez) If no length indicator is given, a `wstring` stores the length in the first two bytes of the string. The maximum size of a `wstring` is 65,535.

Index

Entries in **boldface** are menu commands. Entries in `typewriter face` are functions, methods, variables, keywords, or files.

Symbols

Numerics

A

B

C

D

E

F

G

P

T

U

V

W

SYMANTEC.

Technical Support

For specific technical questions about Symantec C++,
please call our technical experts by choosing one of the three support options below.
For information on Symantec's broad range of service and support programs,
see the Service and Support Solutions section in the User's Guide.

StandardCare Support
503-465-8470 (No charge for 90 days from date of first call.)

PriorityCare 800 or PremiumCare 800 Support
800-927-4014 (Charged on a per-incident or per-year basis.)

PriorityCare 900 Service
900-646-0004 (Charged on a per-minute or per-incident basis.)

Customer Service

For general questions about Symantec products, please call
800-441-7234 (U.S. and Canada) or 503-334-6054.

Symantec Corporation Headquarters
10201 Torre Avenue
Cupertino, California 95014
408-253-9600

05-30-00140