THINK C
Symantec C++

The Professional's Choice

Compiler Guide

Credits

Documentation David Allcott, Bob Foster, Bonnie Hill, Jeff Mattson, Jeanne Munson, Susan Rona, and Cambridge Publications

Development Patrick Beard, Walter Bright, Thomas Emerson, Bob Foster, Greg Howe, Michael Kahl, Darrell LeBlanc, John Micco, Pat Nelson, Daniel Podwall, and Phil Shapiro

Quality Assurance Celso Barriga, Constantine Hantzopoulos, Kevin Irlen, Yuen Li, and Christopher Prinos

Technical Support Celso Barriga, Colen Garoutte-Carson, Rick Hartmann, and Scott Shurr

Project Management David Allcott, Constantine Hantzopoulos, and David Neal

Product Management Steve Levine and Peggy Liu

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

Contents

Welcome 1

Welcome to Symantec C++ for Macintosh. This manual describes the Symantec C++ translator and Symantec's implementation of the C++ language. The Symantec C++ for Macintosh package includes the Symantec C++ compiler and libraries as well as the entire THINK C development environment.

Contents

1 Welcome

If You Are New to the THINK Environment

The *User's Guide* describes how to use the powerful THINK development environment. To learn how to run simple programs with the THINK Project Manager, read "Overview" in the *User's Guide*. The *User's Guide* also contains some simple tutorials in C. You should also look at the two tutorials on C++ in this guide entitled "Hello World++" (Chapter 2) and "Vector" (Chapter 3). Reading the "Overview" chapter from the *User's Guide* and working through these tutorials in this guide is the best way to get started with Symantec C++.

If You Are Learning C++

Read the "Hello World++" tutorial, Chapter 2, and the "Vector" tutorial, Chapter 3, to learn how to run simple programs using Symantec C++. If you're learning 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++

Symantec C++ is a unique development environment for the Macintosh. It features a very fast C++ compiler, an ANSI-conformant C compiler, powerful optimizers, a resource compiler, an extremely fast linker, an integrated debugger, an object inspector, a 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 Macintosh applications, desk accessories, device drivers, and any kind of code resource. 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, and Complex, a library that lets you do mathematic operations with complex numbers.

You can run your program from Symantec C++ 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 THINK Project Manager remains active, so you can examine and edit your source files as you watch your program run.

The Symantec C++ 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 along with Symantec C++, you can edit your source files while you're debugging. In addition, the object inspector lets you browse heap-based objects within your program.

What you need

Symantec C++ requires a hard drive and at least 8 megabytes (8MB) of RAM. Large projects require more memory. Symantec C++ uses temporary memory when it is available and runs in real or virtual memory in both 24- and 32-bit modes.

You can run Symantec C++ on the Macintosh LC, the Macintosh SE series, the Macintosh II series, the Performa series, Powerbooks, Centris, and Quadras. You can also run Symantec C++ on the Power PC under emulation or in emulation mode.

The complete Symantec C++ system takes up about 17MB on your disk, not including your own files. The actual size of your system may be smaller, depending on the kinds of programs you work on. You can customize your installation to use less disk space.

Which System/Finder?

Use the latest System and Finder provided by Apple. Symantec C++ requires System 6 using the Multifinder or System 7.

What Your Package Contains

Your Symantec C++ package consists of seven high-density floppy diskettes, this manual, the *THINK C User's Guide*, and the *Visual Architect and THINK Class Library User's Guide*.

What's in This Manual

The chapters in this manual are: "Tutorial: Hello World++," "Tutorial: Vector," "Using the Symantec C++ Compiler," "Symantec C++ THINK Inspector," "Language Reference," "Compiler Options Reference," and the appendixes "Porting Code," "ARM Conformance," and "Symantec C++ Errors." 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*.
Tutorial: Hello World++	This basic tutorial shows you how to put together an application with Symantec C++.
Tutorial: Vector	This is a tutorial on Symantec C++.
Using the Symantec C++ Compiler	This chapter describes how Symantec C++ compiles your source files. It also tells you how to change language settings, compile code for the 68881/2 coprocessors, and use the global optimizer.
Symantec C++ THINK Inspector	This chapter describes the operation of the Symantec C++ THINK Inspector.
Language Reference	This chapter describes in detail aspects of the Symantec C++ implementation that are not part of the C++ language definition.
Compiler Options Reference	This chapter describes special features and extensions in Symantec C++. It discusses special object types, pragmas, and

	Macintosh-specific extensions to the C++ language.
Porting Code	This appendix describes how to port code from THINK C and MPW C++ to Symantec C++. It also contains hints on how to port from other C++ implementations.
ARM Conformance	Different compilers implement the language in slightly different ways. This appendix discusses implementation-specific issues by referring to *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1990; and *The C++ Programming Language, Second Edition* by Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1992.
Symantec C++ Errors	This appendix describes error messages generated by Symantec C++.

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 at least learning, how to program in C++. If you're just getting started in C++, Symantec C++ 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 more than one thousand Macintosh Toolbox routines. The Toolbox is the set of operating system and user interface routines that make a Macintosh a Macintosh. *THINK Reference* 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. The "Welcome" chapter of the *User's Guide* contains a list of a number of books that describe how to program the Macintosh as well as a list of reference works.

Learning C++

As the popularity of C++ grows, 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. These things are handled very differently in Symantec C++, so when you choose a book, choose one that doesn't dwell on these aspects of programming.

If you're learning C++ from a book, or if you're using Symantec C++ to do course work, be sure to do the "Vector" tutorial (Chapter 3). It shows you how to set up to write and run C++ programs that use the standard C and C++ libraries.

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++.

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 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.

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.

Tutorial: Hello World++ 2

The purpose of the "Hello World++" tutorial is not to teach you to write a fancy program, but to show you how to build an application in Symantec C++ for Macintosh. The program writes the words "hello world" in a window on the screen.

Before You Begin

Be sure to follow the instructions in the *User's Guide*, Chapter 2, "Installation," to install Symantec C++ for Macintosh on your disk.

What You Should Know

You should know how to use the standard file dialog boxes to move around between different folders. If you don't, read the user's manual that came with your Macintosh.

Contents

2 Tutorial: Hello World++

Creating the Project

First you need to launch the THINK Project Manager. To do this, open the folder containing the THINK Project Manager and double-click the THINK Project Manager icon.

A dialog box asks you to open a project.

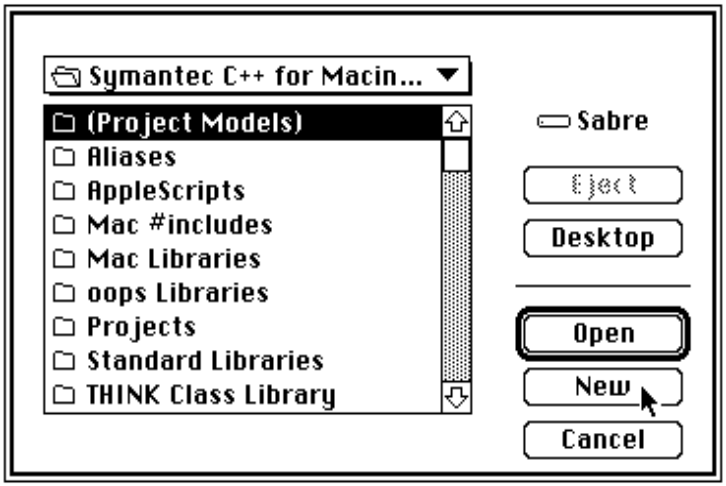


Figure 2-1 Opening a project

Since you're creating a new project, click the New button. You'll see another dialog box that lets you select from various types of projects. Since the purpose of this tutorial is to demonstrate how to build a simple project from scratch, select Empty Project from the list and click the Create button. Normally you would use one of the built-in project models when creating new projects, or a custom project model of your own design.

Now a dialog box asks you to name your new project and choose where it will be stored on disk. Move back to your `Development` folder, name the project `Hello++`, and click Save.

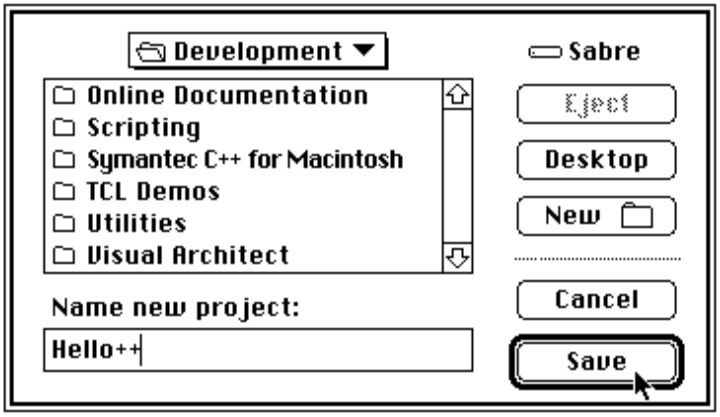


Figure 2-2 Naming your new project

Warning

You can give your project a different name, or store it somewhere other than in the `Development` folder, but remember that your dialog boxes won't match the figures in this chapter.

Note

Generally, you won't store projects in the THINK tree (the folder in which THINK Project Manager and its subfolders reside). In particular, despite its name, you should not store projects in the `Projects` folder. This folder is for storing aliases to frequently used projects that you want to appear in the **Switch to Project** submenu.

THINK Project Manager creates a new folder named `Hello++ f`, and inside it a new project document named `Hello++.π`. Then THINK Project Manager displays a new, empty project window:

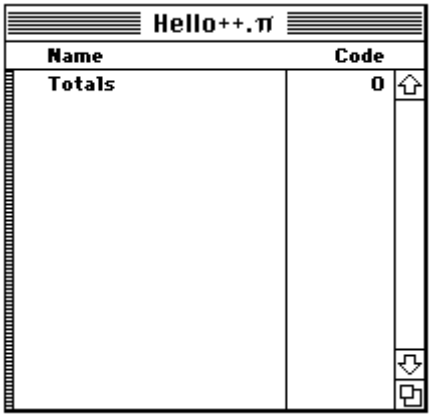


Figure 2-3 The project window for a new, empty project document

The Name column shows the names of the source files and libraries in your project. The Code column displays their sizes in bytes.

Creating the Source File

Now you're ready to create your source file. Choose **New** from the **File** menu to bring up an empty, untitled editing window.

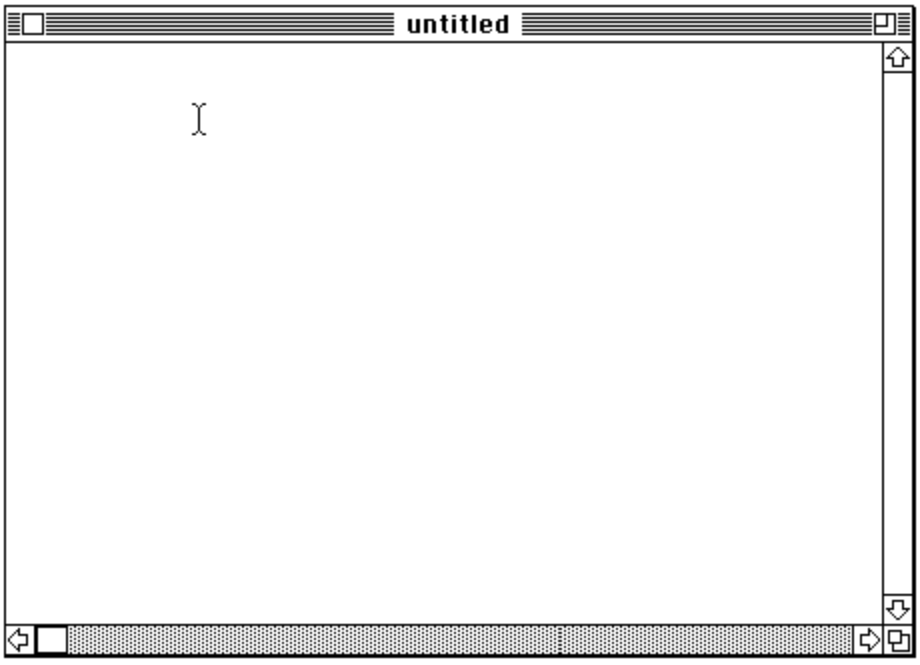


Figure 2-4 Untitled window

Type the following program into the editor window (you don't need to type in the comments if you're in a hurry):

```
/*****
* hello.cp
*
* The hello world program for Symantec C++ for
* Macintosh
*
*****/
#include <iostream.h>
void main()
{
cout << "hello world!\n";
}
```

The THINK Project Manager text editor works like most other text editors on the Macintosh. You can drag to select a range of text or double-click to select words. You can also triple-click to select an entire line. If you have a keyboard with arrow keys, you can use them to move around your file.

The text editor does not wrap text when you type past the right edge of the window. Use the horizontal scroll bar at the bottom of the window to view any text that goes beyond the right edge.

After you type the program, choose **Save As** from the **File** menu to save it. You get a dialog box like the one below. Name the file name `hello.cp` and click Save.

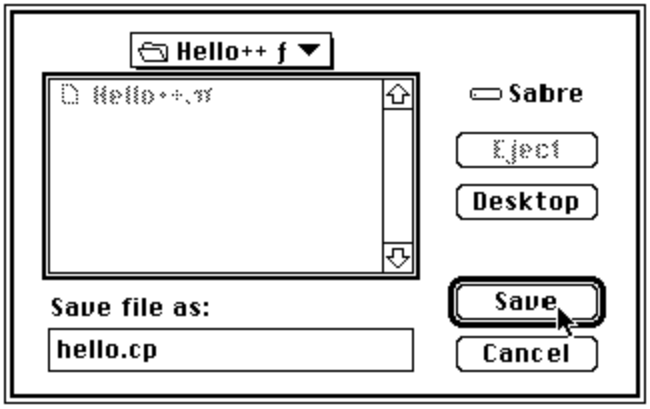


Figure 2-5 Save As dialog box

You can edit any text file with the THINK Project Manager editor, not just those ending in `.cp`.

Warning

Make sure the name of your file is `hello.cp`, not `hello.c`. By default, THINK Project Manager uses the C++ translator to compile `.cp` and `.cpp` files, but uses the C translator to compile `.c` files. The program you just typed will not compile as a C program.

Compiling the Source File

Now you're ready to compile your source file. Choose **Compile** from the **Source** menu. THINK Project Manager displays a dialog box that shows how many lines have been compiled.

When THINK Project Manager compiles a source file, it adds its name and size to the project window. Your project window should now look like this:

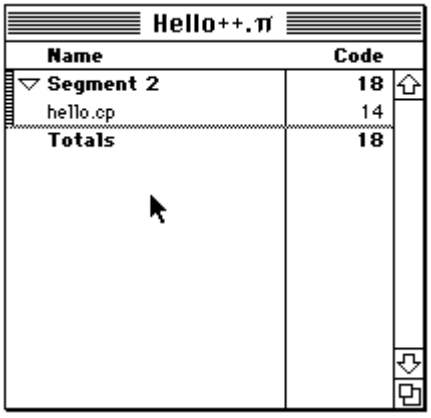


Figure 2-6 Project window

THINK Project Manager keeps the object code for your source files in the project document.

Did You Get an Error?

If you made a mistake typing the program, THINK Project Manager displays an error message in a window called Compile Errors. The message may say `syntax error`. In this small program, about the only syntax error you can make is forgetting a quotation mark, a parenthesis, or a semicolon.

Double-click the error message to find the source of the error. THINK Project Manager selects the line with the error. Correct the error and look over your program to make sure everything else is correct. Then choose **Compile** from the **Source** menu.

The following error message indicates that THINK Project Manager wasn't able to find the include file `iostream.h`.

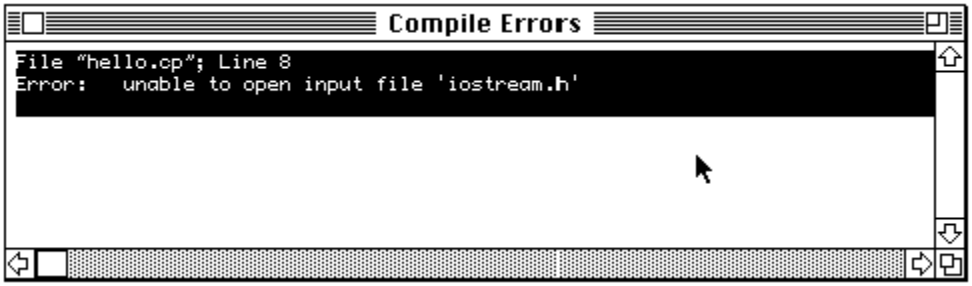


Figure 2-7 Sample Compile Errors window

The THINK Project Manager can't find the include files if the `Standard Libraries` folder isn't in the `Symantec C++ for Macintosh` folder, or if you've moved the application from its installed location. If you have moved the THINK Project Manager application, move it back to the `Symantec C++ for Macintosh` folder and start over. Quit THINK Project Manager and move the `Hello++ ƒ` folder to the Trash. Then look in the *User's Guide*, Chapter 2, "Installation," to make sure you installed Symantec C++ for Macintosh correctly. Once you're sure everything is OK, start from the beginning of this chapter.

Warning

Drag the `Hello++ ƒ` folder into the Trash only if you're starting over. If you didn't get the "Unable to open input file" error message, proceed.

Adding the Libraries

If you tried to run your program now, you'd get link errors because the project doesn't know where the `<<` operator, the `cout` stream variable, and other essential functions are defined.

Next, you need to add three libraries to the project. The CPlusLib library is required for all C++ projects that produce applications. The ANSI++ library is required only for non-Macintosh applications. The IOStreams library defines the `<<` operator and the `cout` stream variable and all the standard streams library routines. To add the libraries, choose **Add Files** from the **Source** menu. You'll see a dialog box like the one in Figure 2-8.

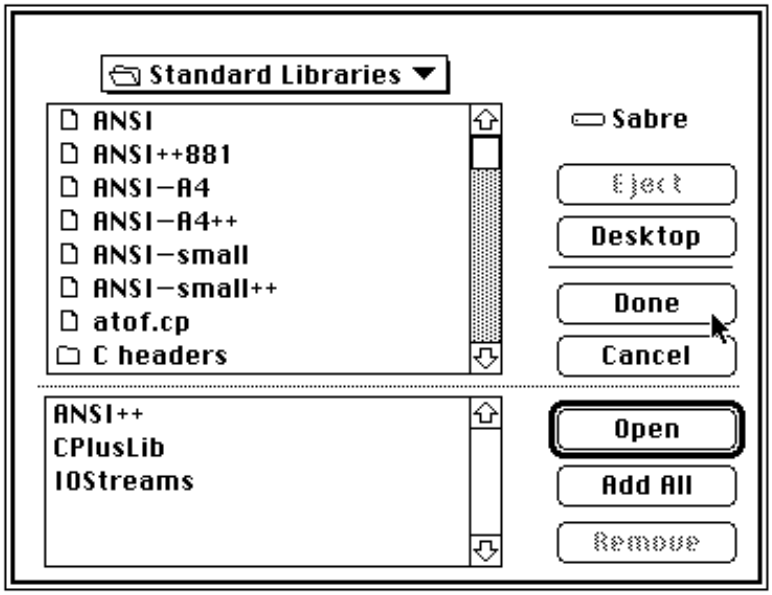


Figure 2-8 Standard libraries

The top list is used to display the source files and libraries in the folder you're currently in. The bottom list shows the files THINK Project Manager will add to your project when you click Done. The Add and Add All buttons move files into the bottom list. The Remove button removes files from the bottom list.

Open the folder called `Standard Libraries`, in the `Symantec C++ for Macintosh` folder. This folder contains all the libraries for ANSI and Unix compatibility, including the ANSI++, CPlusLib, and IOStreams libraries. Select `ANSI++` and click the Add button. The file moves to the bottom list. Repeat this procedure for the CPlusLib and IOStreams libraries. Make sure that the dialog looks like the one in Figure 2-8, then click the Done button.

THINK Project Manager adds the libraries ANSI++, CPlusLib, and IOStreams to the project window. Your project window should look like this:

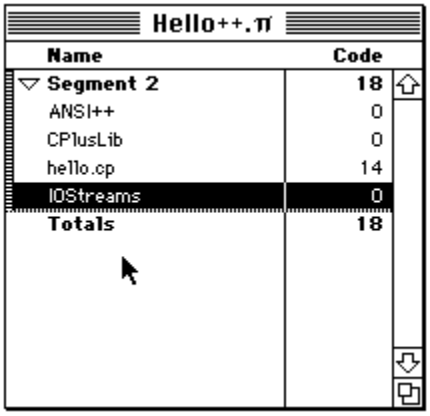


Figure 2-9 Project window

Now that the libraries have been added to the project, you must move some of them into different segments. Select `ANSI++` and drag it below the gray line. Segment 3 is created containing the ANSI++ library. Create another segment for the IOStreams library. Your project now contains three segments.

The project window should look like this:

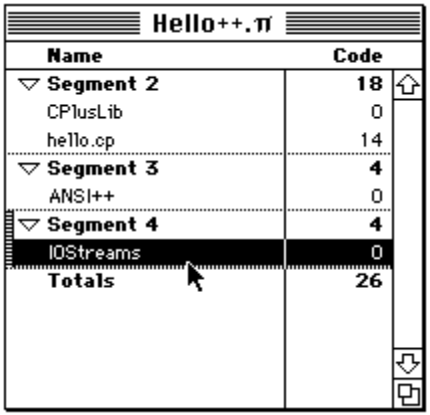


Figure 2-10 Creating project segments

The object size for each of the libraries is zero, because the THINK Project Manager doesn't load a library's code until you need it. As a result, you can add several libraries without waiting for them to load.

THINK Project Manager loads a library automatically when you run the project. Another way to load a library is to click its name in the project window and then choose **Load Library** from the **Source** menu. For this example, let THINK Project Manager load it for you.

Note

Each segment in a project must contain less than 32K of object code. When loaded, the ANSI++ and IOStreams libraries take up close to 32 kilobytes (32KB) each, which is why you must put them in separate segments. If you try to run a project that is not segmented properly, the THINK Project Manager, by default, asks if you want it to auto-segment your project. You can have the THINK Project Manager automatically segment your projects by setting this option in the Preferences page of the **THINK Project Manager options** dialog. In this tutorial, however, you learn how to segment the project manually.

Running the Project

Everything is now set to run the project. The source file is in the project window along with the libraries you'll be using. Choose **Run** from the **Project** menu.

The THINK Project Manager notices that the libraries need to be loaded, so it opens a dialog box asking if you want to bring the project up to date:

Figure 2-11 Update dialog box

Click the Update button. The THINK Project Manager goes to disk to load the code for the CPlusLib, ANSI++, and IOStreams libraries. A little time may be needed to load the libraries; but once they're loaded into the project, THINK Project Manager doesn't need to load them again.

Any time you choose to run your project and the THINK Project Manager notices you've made changes (added libraries or source

files, or edited source files), it asks if you want to update the project. If you say yes, it compiles the new or changed files and loads the new libraries. Because this program uses the IOStreams library, everything sent to `cout` goes to a console window. A console window is a Macintosh window that behaves like a simple display terminal, the kind of terminal that many MS-DOS and Unix computers use. You'll see the `hello world!` string at the bottom of this window.

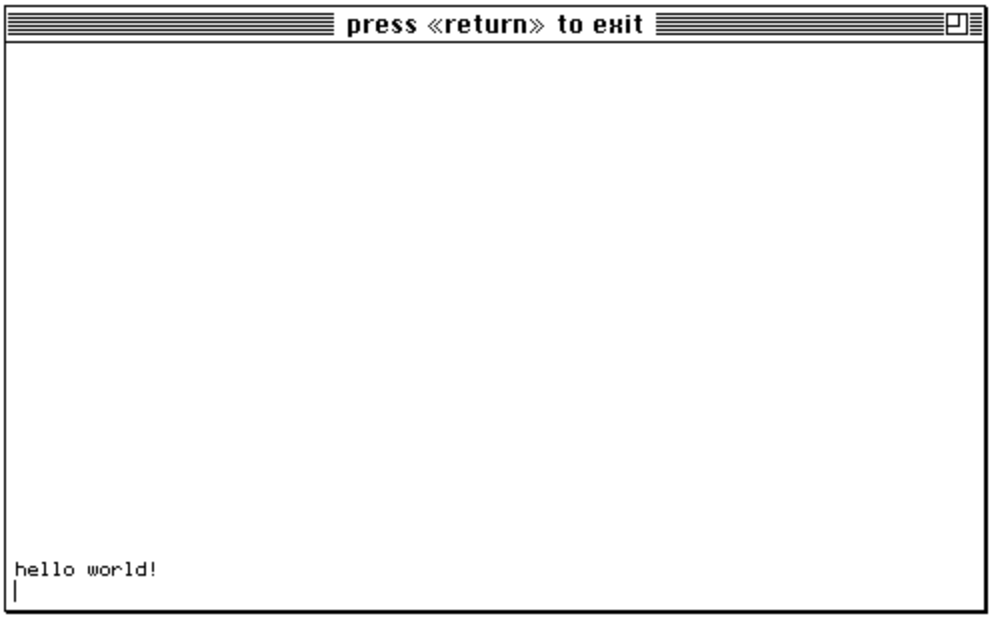


Figure 2-12 Console window

To exit the program, press Return or choose **Quit** from the **File** menu.

Creating the Application

As you develop a large application, you make changes to your source files. Each time you run your project, THINK Project Manager will recompile only those files that have changed. When you're ready to turn your project into a stand-alone double-clickable application, choose **Build Application** from the **Project** menu.

A dialog box asks you to name your application. Name it `hello++ appl`. Leave the Smart Link check box checked. This option tells THINK Project Manager to make your application as small as possible. Be sure to move to the `Hello++ ƒ` folder before clicking Save.

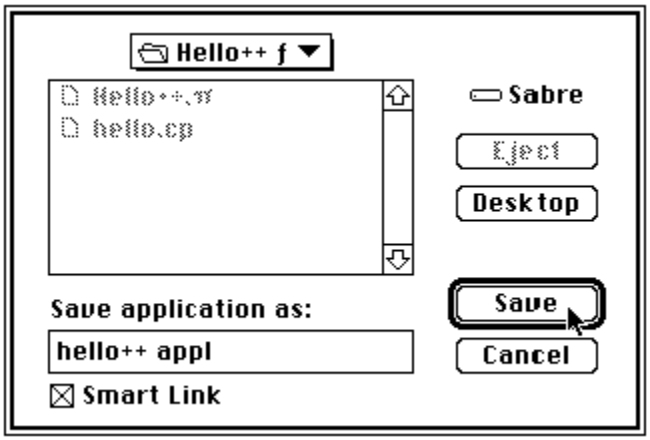


Figure 2-13 Build Application dialog box

THINK Project Manager puts up a dialog box telling you it's linking your application. When it's finished, the application is in the folder you chose.

To run your application, bring up the window with the folder your application is in. Double-click your application and watch it run.

Where to Go Next

The tutorial in the next chapter is a more elaborate example of building an application with Symantec C++ for Macintosh. It describes how THINK Project Manager reports errors when you compile and link, and it shows you some advanced features of the THINK Project Manager editor.

2 Tutorial: Hello World++

Tutorial: Vector 3

Vector shows you how to use some unique features of Symantec C++. Vector is a small application that uses templates to implement vectors and a sorting function.

Before You Begin

Be sure that you've installed Symantec C++ correctly. The Vector tutorial should be in the `Vector f` folder in the `Demos` folder in your `Development` folder.

What You Should Know

Before you try this tutorial, you should know how to use the THINK Project Manager and the THINK Debugger. If you're not familiar with them, work through the tutorials in the *User's Guide*.

This tutorial shows you how to use templates with Symantec C++, but it does not teach you about templates in general. To learn about templates, see *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup and *The C++ Programming Language, Second Edition* by Bjarne Stroustrup.

Contents

About the Vector Project

The Vector project is an application that uses inline functions and templates to give you an idea of how they work in Symantec C++. It uses templates to implement a vector (array) class of any type, finds the maximum value in a vector, and sorts any kind of vector. The Vector application displays its output, including the results of two inline functions, on the console.

Before you proceed with this tutorial, run the Vector project without the debugger to familiarize yourself with the way the vector works. Make sure that the **Use Debugger** option in the **Project** menu is not checked, and choose **Run** from the **Project** menu.

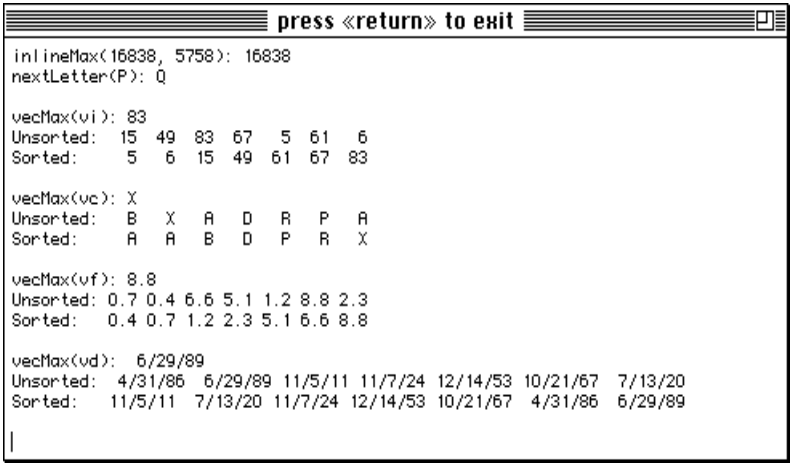


Figure 3-1 Running the Vector application

In this tutorial, you run the Vector application several times. After it runs, notice that the title of the window changes from “console” to “press «return» to exit.” To exit the Vector application, press Return, or choose **Quit** from the **File** menu when the console window is active. If the source-level debugger is active, you can also choose **ExitToShell** from the **Debug** menu.

The Always save session option in the Debugging page of the **THINK Project Manager options** dialog is turned off in this project. Since you’ll be running the project several times, it is more convenient if you don’t have breakpoints left over from previous runs.

Most of the procedures in this tutorial start from the file `main.cp`, so you may want to look at that title before you continue.

Debugging Inline Functions

To cut down on the overhead of function calls for small functions, C++ provides inline functions. Instead of generating code for a function call, Symantec C++ generates the code for the function where the function call would otherwise appear. In C++ there are two ways to declare inline functions. One way, for global functions, is to use the `inline` specifier. The second way is to provide the definition (not just a declaration) for a member function in a class declaration.

In the Vector application, the function `inlineMax()` in the file `main.cp` and the function `nextLetter()` in the file `next.h` are global inline functions. In the template class `vector` in the file `vector.h`, the constructor, destructor, and member function `size()` are inline functions because the definitions are provided in the class declaration.

Although inline functions behave syntactically like normal functions, you can't debug them because the compiler doesn't generate a function call for them. That means that you can't set a breakpoint in an inline function and that you can't step into an inline function.

This option is on in the Vector project.

To debug inline functions, turn on the Use function calls for inlines option in the Debugging page of the **THINK Project Manager options** dialog. When this option is on, Symantec C++ generates normal function calls for inline functions.

Keep in mind that the inline specifier is a hint to the compiler that it should try to generate the code directly instead of creating a function. Just as not every variable declared `register` necessarily ends up in a register, a function declared `inline` may not actually be an inline function.

Here are examples of debugging inline functions.

If the function is in a source file, set a breakpoint in the inline function as you would for any other function. Choose **Run** from the **Project** menu. When the debugger windows appear, scroll up until you see the `inlineMax()` function. You can set a breakpoint in it the way you usually do.

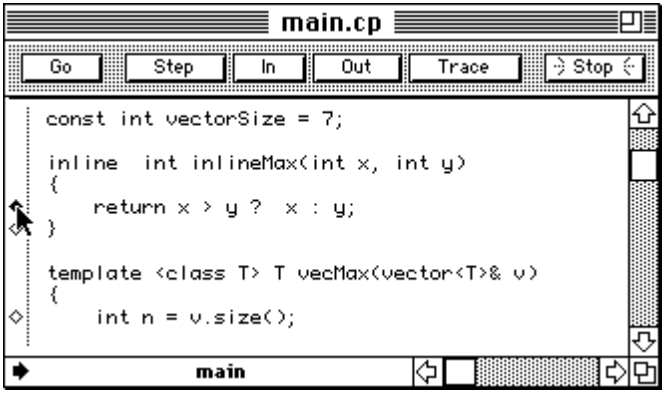


Figure 3-2 Setting a breakpoint in inlineMax()

If the inline function is in a header file, use the pop-up menu from the Source window to open it. To set a breakpoint in the function `nextLetter()`, hold down the Option key as you click the Source window's title bar, and choose **next.h** from the pop-up menu.

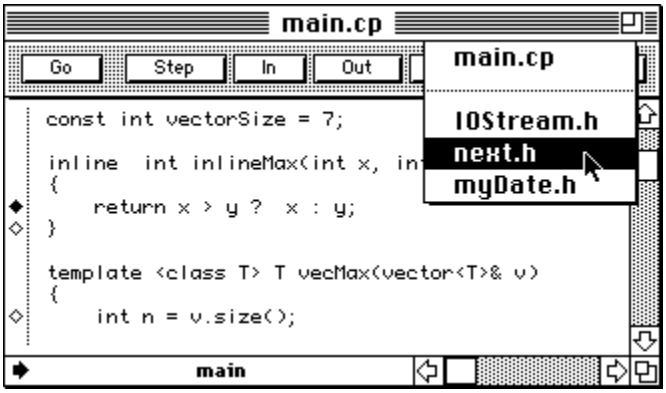


Figure 3-3 Opening a header file in the Source window

The header file appears in the Source window. You can now set a breakpoint in the `nextLetter()` function.

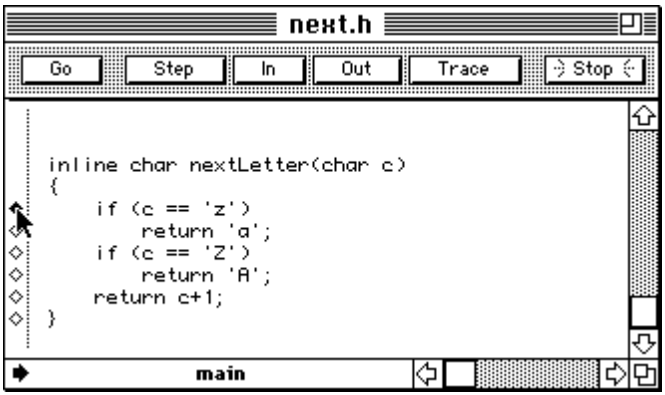


Figure 3-4 Setting a breakpoint in nextLetter()

If the Use function calls for inlines option is off, you don't see any breakpoint diamonds next to the `inlineMax()` function, and the `next.h` file isn't available in the pop-up menu because it doesn't generate any code.

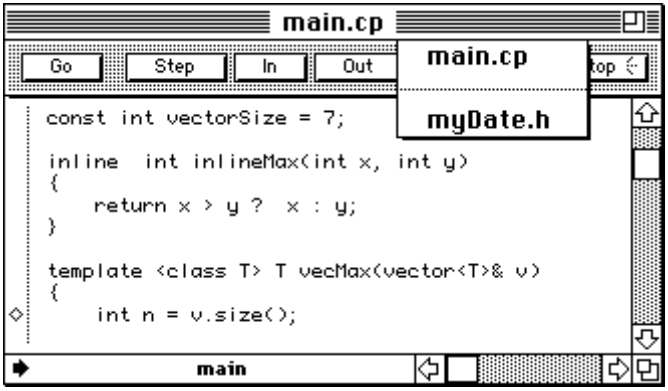


Figure 3-5 Use function calls for inlines option turned off

Using and Debugging Templates

The template mechanism in C++ lets you define container classes and generic functions without giving up type checking. The Vector project provides examples of both a container class (the vector class) and a generic function (the sorting routine).

As the names imply, template functions and template classes are not actual functions and classes. Rather, they are schematics for building real functions and classes for types that you specify. Because templates behave differently from normal functions and classes, you need to understand how Symantec C++ compiles and generates debugging information for them.

This section introduces some useful techniques for using and debugging templates in Symantec C++.

Instantiating templates

The file `instance.cp`*, which is located in the same folder as the Vector project, contains the code from this section so you can experiment with it.*

Keep in mind that the compiler never generates code for template definitions. Consider this trivial function template:

```
template<class T> T sq(T v)
{
   return v * v;
}
```

Symantec C++ compiles the template, but it doesn't generate code for it. To generate code, you need to instantiate the function. There are two ways to do this: Use the template, or a `#pragma` directive.

This is how you instantiate a template function `by use`:

```
void byUse()
{
   int i = 5;
   float f = 3.14;

   i = sq(i);
   f = sq(f);
}
```

In this example, the compiler instantiates two versions of the function `sq()`. One version takes an `int` argument and the other takes a `float` argument.

This is how you use the #pragma directive:

```
#pragma template sq(int)
#pragma template sq(float)
```

These two directives instantiate the int and float versions of the function sq().

Instantiation is similar for class templates. The only difference is the syntax of the #pragma directive:

```
#pragma template tVector<int> // instantiate an
                              // int version of
                              // tVector
```

This tutorial shows you two ways to use templates. The first, called simple templates, is a straightforward instantiation by use. The second employs instantiation files and the template pragma to instantiate specific versions of template functions.

A simple template is a template that you include through a header file or that you write in your code. The advantage of simple templates is that you don't have to create special files for each instance of a template function. The disadvantage is that you cannot debug every instance of a template function or class.

Instantiation files use a more elaborate header file and source file arrangement. This produces one file in the project for each instance of a template function or class, all of which you can debug.

Templates and debugging information

To understand the difference between simple templates and instantiation files, it helps to know how the Symantec C++ compiler generates debugging information for template functions and member functions.

For source files, Symantec C++ generates debugging information for each function or member function for which source code is available. This works well for normal functions and member

functions, since there is always a one-to-one correspondence between the source code and the object code, even for overloaded functions. Consider these two functions:

```
int max(int a, int b)
{
   return a > b ? a : b;
}

char max(char a, char b)
{
   return a > b ? a : b;
}
```

The Symantec C++ compiler generates object code and debugging information for the function `max(int,int)` and for the function `max(char,char)`.

Now consider this template function:

```
template<class T> T tMax(T a, T b)
{
   return a > b ? a : b;
}

void foo(int i, int j, char x, char y)
{
   int n;
   char c;

   n = tMax(i, j)
   c = tMax(x, y);
}
```

The Symantec C++ compiler doesn't generate object code or debugging information for the template itself. It generates code only when the function is instantiated. In the example above, Symantec C++ generates object code for the functions `tMax(int,int)` and `tMax(char,char)`. Since there is source code available, the compiler also generates debugging information. But since the debugger requires a one-to-one correspondence between source code and object code, the debugging information applies to only one instance of the `tMax()` function. Unfortunately, you cannot tell which instance has debugging information. If you set a breakpoint in `tMax()`, you might stop in `tMax(char,char)` or in `tMax(int,int)`.

Debugging simple templates

In the Vector application, the template function `vecMax()` in the file `main.cp` uses the simple template approach. This is how `vecMax()` appears in `main.cp`:

```
template <class T> T vecMax(vector<T>& v)
{
   int n = v.size();
   T max = v[0];

   for (int i = 1; i < n; i++)
      if (v[i] > max)
         max = v[i];

   return max;
}
```

The function takes a vector of a particular type `T` as an argument and returns the largest element in the vector. This function works for any type for which the greater-than operator is defined.

*You can use the **Disassemble** command in the **Source** menu to examine the code that Symantec C++ generates for each version.*

In the function `main()`, there are four calls to `vecMax()` for the built-in types `int`, `char`, and `float`, and the user-defined type `myDate`. The appearance of each call to `vecMax()` creates a new instance of the function. When you compile the file `main.cp`, Symantec C++ generates code for four different versions of `vecMax()`. As explained above, the compiler generates debugging information for only one of those instances.

To see what that looks like, make sure that the Use Debugger option is on, and choose **Run** from the **Project** menu.

Scroll toward the beginning of the file `main.cp` and set a breakpoint in the template function `vecMax()` like this:

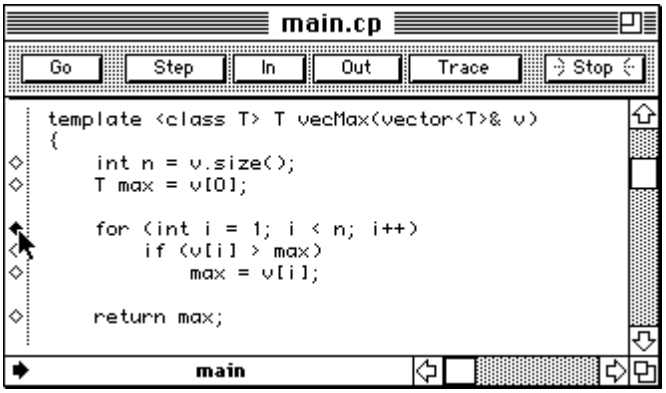


Figure 3-6 Setting a breakpoint in vecMax()

Now click the Go button and notice where execution stops:

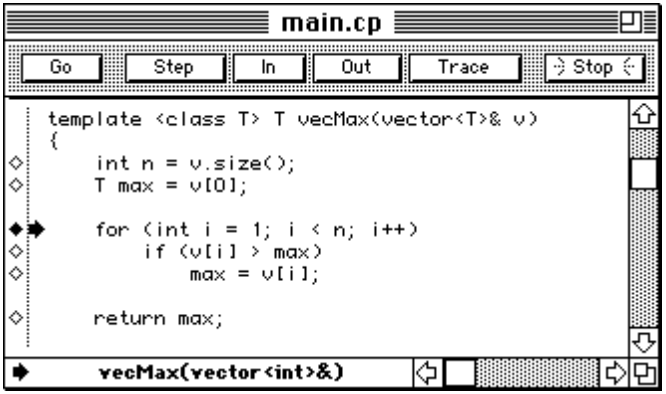


Figure 3-7 Breaking inside vecMax()

From the current function display at the bottom of the Source window, you can see that the debugger stopped execution at the `vecMax(vector<int>&)` instance of the template function. Click the Go button again, and notice that the program does not stop this time.

Note

The current version of Symantec C++ generates debugging information for the first instantiation of a template function. However, you should not rely on this behavior; it may change in the future.

As you can tell from the output in the console window, even though Symantec C++ generated debugging information for only one instance of the `vecMax()` function, it did generate code for the other three versions.

If you are writing an uncomplicated template function such as `vecMax()`, there is nothing wrong with using the simple template technique. For such functions it's unlikely that you need the source-level debugger at all. But if you are writing a more complex template function or a template class, you may need to be able to debug every instance or a particular instance of the template. To do this, you need to use the instantiation file technique.

Using template instantiation files

The template instantiation file technique forces the Symantec C++ compiler to generate debugging information for each instance of a template function or class. The Vector application uses this technique for the `vector` class and for the `selection()` function, which implements a selection sort.

First, create a file for the template class declaration or the template function declaration. In the Vector project, the `vector` class declaration is in the file `vector.h`. Include this file in any other source file that needs the declaration of your template class. The `selection()` template function declaration is in the file `selection.h`. Include this file in any source file that calls for the template function to provide a function prototype for `selection()`.

Next, create an implementation file for the definitions of the template member functions and the template function. By convention, the name of this file is the name of the class or function. For the `vector` class, the implementation file is the file `vector.cp`. It contains the definitions of all the template functions that are not defined as inline functions. For the `selection()` function, the implementation file is the file `selection.cp`. Do not add this file to the project.

Finally, create an instantiation file for each instance of the template class or template function. By convention, the name of this file follows the same format as `#pragma template` directive. The instantiation file for the `int` version of the `vector` class is named `vector<int>.cp`. The instantiation file for the `float` version of the `selection()` function is named `selection(float).cp`. The contents of the instantiation file look like this:

```
#include "vector.cp"
#pragma template_access public
#pragma template vector<char>
```

The `#include` statement brings in the implementation file. The `#pragma template_access public` directive ensures that the scope of the instantiation of the class and its member functions is public. If you leave out this directive, the scope is static, and the class and its member functions are available to other files in the project. As you read earlier, the directive `#pragma template vector<char>` asks the Symantec C++ compiler to instantiate a `char` version of the `vector` class.

The instantiation file for template functions looks the same. The only difference is the function syntax for `#pragma template`:

```
#include "selection.cp"
#pragma template_access public
#pragma template selection(vector<float>)
```

Add the instantiation file to the project.

Note

The other files in the project must be compiled with the `#pragma template_access extern` for this technique to work.

As you can see in the Vector project, there are four instantiation files for the `vector` class and four instantiation files for the Vector class's `friend` function.

Debugging with instantiation files

When you use the template instantiation file technique, Symantec C++ generates object code for only one instance of the template. When the debugging option is on, it generates debugging information only for that instance, so the debugger is able to maintain a one-to-one relationship between object code and source code.

Since the instantiation file in the project doesn't contain code, use the pop-up menu from the debugger's Source window to reach the implementation file that the instantiation file includes.

Here's how to set a breakpoint in the `char` version of the `selection()` function.

Choose **Run** from the **Project** menu to run the Vector application. Click the project window to make it active, or choose **Vector.π** from the debugger's **Windows** menu.

Click the file name `selection(char).cp` in the Project window, and choose **Debug** from the **Source** menu. The file appears in the debugger's Source window.

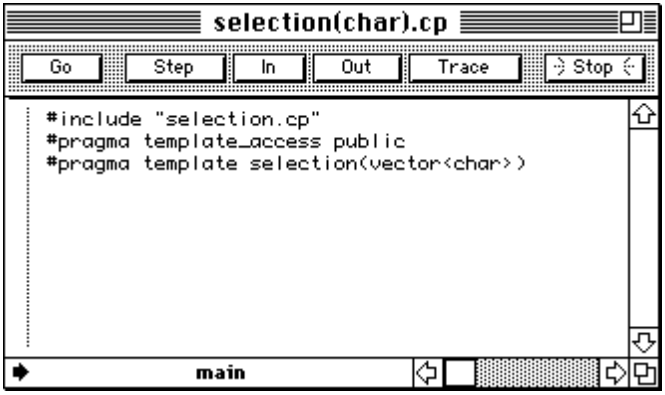


Figure 3-8 Instantiation file in the Source window

To see the code in the included implementation file `selection.cp`, hold down the Option key as you click the Source window's title bar. Choose the implementation file from the pop-up menu.

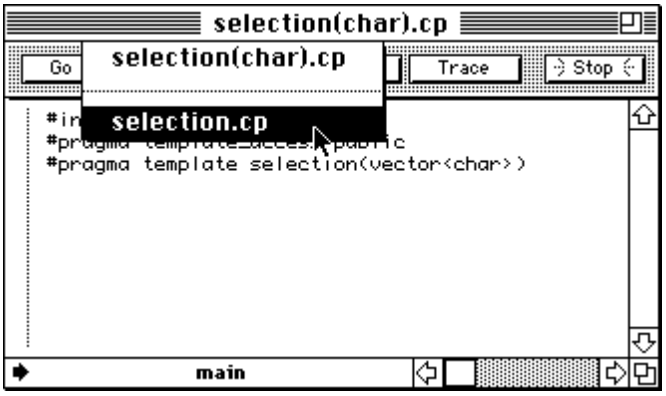
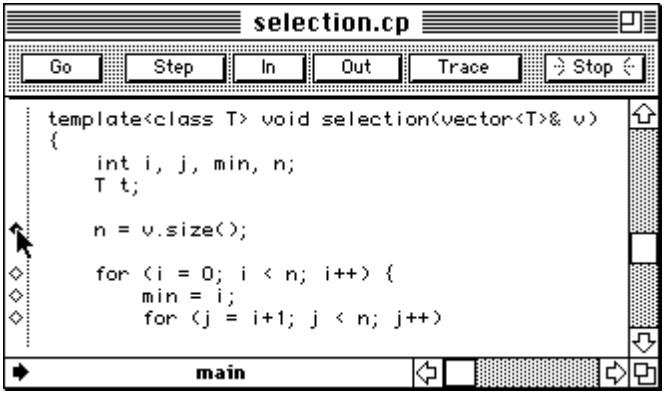


Figure 3-9 Choosing the implementation file

The debugger's Source window now shows the source code in the implementation file. Set a breakpoint in the function `selection()`.



breakpoint

Figure 3-10 Setting a breakpoint in the implementation file

Click the Source window's Go button. Note that execution does not stop in the `int` version of `selection()`. It stops only for the `char` version of `selection()`:

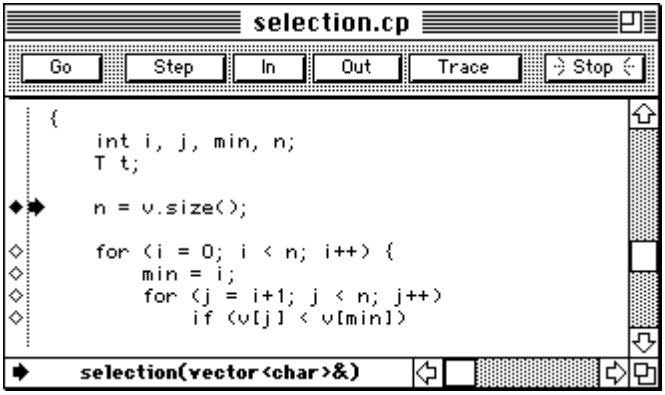


Figure 3-11 Stopping in the char version of selection()

You can use the same steps to set breakpoints for other instantiations of `selection()` or to set breakpoints in the `vector` class member functions for specific types.

What to Do Next

This tutorial showed you some basic techniques for working with inline functions and templates in Symantec C++. If you're just learning C++, you might find the Vector program useful for trying out different things.

Create wrapping subscripts

The vector class shows how to write a subscript operator. Rewrite it so that the subscripts wrap around when they're out of range. For example, if the vector object `foo` contains ten items, the in-range subscript references are `foo[0]` to `foo[9]`. If you use any out-of-range subscript, like `foo[10]`, the current subscript operator always returns the first element in the vector, `foo[0]`. But if the subscript operator wrapped, it would return `foo[0]` for `foo[10]`, `foo[1]` for `foo[11]`, `foo[2]` for `foo[12]`, and so on.

Add new methods to myDate

- A `>>` operator that reads in a date from the console or a file. First, write the function to read dates written such as `05/04/95`. Then, extend it to read dates written such as `Apr. 28, 1996`. Declare the function this way:

```
friend ostream& operator>>
    (ostream& s, const myDate& d);
```

- A `>` operator that returns whether one date is later than another. For example, June 30, 1995 is later than December 13, 1994. Declare the function this way:

```
friend int operator>
    (const myDate& d1, const myDate& d2);
```

- A `+` operator that adds a number of days to a date and returns a new date. For example, 21 days + December 12, 1994 = January 2, 1995. You have to write two functions: one that takes the date first and the number of days second, and another that takes the number of days first and the date second. These functions need to take into account leap years and the number of days in each month. Declare one of the functions this way:

```
friend myDate operator+
    (const myDate& date,
     const int& days);
```

- A `-` operator that subtracts one date from another and returns a number of days. For example, March 15, 1995 - January 25, 1995 = 49 days, and July 6, 1995 - August 10, 1995 = -35 days. This function needs to take into account leap years and the number of days in each month. Declare the function this way:

```
friend int operator-
    (const myDate& d1, const myDate& d2);
```

Write a new sort function

Use the `selection()` function in the file `selection.cp` as a starting point for a different kind of sort. You can find sorting algorithms in many computer science textbooks and tutorials.

Create a new class and sort it

Write your own class; for example classes for strings, times, or people's addresses; create a vector from it and sort it.

Change the vecMax() function into a member function

Make the `vecMax()` function a member function of vector. Remember, the new member function won't take a vector object as an argument. Instead, call the function like this:

```
i = myVector.max();
```

Create a template function

The code that prints out the vector maximum, the unsorted vector, and the sorted vector is repeated four times in `main()`. Write a template function that does its work.

3 Tutorial: Vector

Using the Symantec C++ Compiler 4

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 Symantec C++, including how to compile source files, how to use precompiled headers, and how to set options that affect the way Symantec C++ compiles your source files. This chapter also lightly touches on how Symantec C++ complies with the Draft ANSI standard for the C++ language.

Contents

4 Using the Symantec C++ Compiler

Compiling Source Files

Unlike traditional compilers, Symantec C++ doesn't generate separate object files from your source files. Instead, Symantec C++ puts 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

Source files are your program files. Object code is the machine language that the Symantec C++ compiler generates from your source files.

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 C++ editor. Save your file in the same folder as the project document. Make sure that the file name ends in `.cp` or `.cpp`. By default, Symantec C++ only compiles files that end in `.cp` or `.cpp`.

Note

You can change the file extensions that the THINK Project Manager uses to choose translators.

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

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 **Source** 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 `.cp` or `.cpp`). Next, choose **Check Syntax** from the **Source** 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++ detects an error in your source file, it opens the Compile Errors window. If there is more than one error, Symantec C++ reacts according to the settings you chose in the **Options** dialog. One setting you can choose is to have all errors listed in the Compile Errors window. If you double-click an error in this window, the source file that contains the error opens in an Editor window with the line that contains the error highlighted.

Error reporting

The Error reporting radio button cluster in the Debugging page of the Symantec C++ **Options** dialog lets you choose how Symantec C++ reports compiler 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 7 for more information on error reporting.

Note

If you are using the auto-make facility, the compiler will not stop after an error, nor will it stop after any errors in the source file.

Precompiled Headers

Symantec C++ 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 do text header files.

Another benefit of precompiled headers is that they make the debugging information stored with the project, or in separate `XSYM` files, much smaller.

`MacHeaders++` is a precompiled header containing the most common declarations you use for writing Macintosh programs. It can be found in the `Mac #includes` folder. If you include `<MacHeaders++>` in the project prefix for Symantec C++, Symantec C++ automatically includes `MacHeaders++` in the files in your project. 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.) To edit the project prefix, see "Creating your own precompiled header" later in this chapter.

Note

The project prefix for Symantec C++ contains the line `#include <MacHeaders++>` by default.

The `MacHeaders++` file contains these files:

```
AEObjects.h          AEPackObject.h
AERegistry.h         BDC.h
Controls.h           Desk.h
Devices.h            Dialogs.h
DiskInit.h           Errors.h
Events.h             Files.h
Fonts.h              GestaltEqu.h
Lists.h              Memory.h
Menus.h              Notification.h
OSEvents.h           OSUtils.h
Packages.h           PrintTraps.h
QuickDraw.h          Resources.h
Scrap.h              Script.h
SegLoad.h            StandardFile.h
TextEdit.h           Timer.h
ToolUtils.h          Types.h
Windows.h            pascal.h
```

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

```
ADSP.h                  AIFF.h
Aliases.h               AppleTalk.h
Balloons.h              CommResources.h
Connections.h           ConnectionTools.h
CRMSerialDevices.h      CTBUtilities.h
DatabaseAccess.h        DeskBus.h
Disks.h                 Editions.h
ENET.h                  EPPC.h
FileTransfers.h         FileTransferTools.h
Finder.h                FixMath.h
Folders.h               Graf3D.h
HyperXCmd.h             Icons.h
Language.h              MIDI.h
Palettes.h              Picker.h
PictUtil.h              Power.h
PPCToolBox.h            Printing.h
QDOffScreen.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
Traps.h                 Values.h
Video.h
```

Usually, you use the built-in MacHeaders++. You can, however, change the default MacHeaders++ file or make your own precompiled headers.

Customizing the MacHeaders++ file

You might find that in the kinds of programs you write, you frequently refer to a header file that is not already in MacHeaders++. Or, MacHeaders++ might include some files you never use. You can customize the MacHeaders++ file to suit the kinds of programs you write, in the following way:

1. Find the file `Mac #includes.cpp` in the `Mac #includes` folder. Duplicate it and give it a new name, such as `My #includes.cpp`. Open the duplicate with the editor.

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** from the **Source** menu. After Symantec C++ precompiles the file, save it as `MacHeaders++`. (Precompiled files don't go into the project.) The best place for `MacHeaders++` is in the `Mac #includes` folder, but you can save it anywhere in the THINK Project Manager tree.

The auto-make facility marks the files in the current project for recompilation if you change `MacHeaders++`. To let other projects know that `MacHeaders++` has changed, use the **Make** command in the **Source** menu. Click the Use Disk button to mark all the files affected, and then click the Make button to recompile them.

When you add and remove files from `MacHeaders++`, keep this dependency in mind: You cannot `#include` both `LoMem.h` and `SysEqu.h`.

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 in building your precompiled header.

3. Choose the **Precompile** command from the **Source** menu. When Symantec C++ is through precompiling, it asks you to name the file.

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 <MacHeaders++>` in the project prefix, you can't explicitly include any other precompiled header. (A prefix isn't used when precompiling.)

If you don't `#include` any precompiled headers in the project prefix, you can use several different precompiled headers for different parts of your program. You can still explicitly include `MacHeaders++` if you want to use it in certain 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 <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 **Source** menu. It runs the code in the frontmost window through the Symantec C++ preprocessor and displays the result in a new window. The preprocessor expands your macros, includes the contents of your `#include` files, and 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.

Note

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 **Source** 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't disassemble a file that you can't compile.

Generating a link map

Symantec C++ can write a link map for your application. The link map lists your project's segments, including one for global data. For each function (or global variable) in a segment, the map lists its name, its position in the segment, and the file it is defined in. To generate a link map, turn on the Generate link map option in the THINK Project Manager's **Preferences** dialog box.

Symantec C++ creates the link map only when you use the **Build Applications** command. The name of the map is the name of the project with `.map` appended. For example, the link map for the Bullseye.π project is `Bullseye.π.map`. The THINK Project Manager places the link map in the project folder and erases any other link map in the folder.

This is an excerpt from the global data section of a link map:

```
Segment"%GlobalData" size=$000190
   widthMenu       -$000108(A5)file="bullMenus.cp"
   editMenu        -$000104(A5)
   fileMenu        -$000100(A5)
   appleMenu       -$0000FC(A5)
   windowBounds    -$0000F8(A5)file="bullWindow.cp"
   circleStart     -$0000F0(A5)
   width           -$0000E8(A5)
   dragRect        -$0000DA(A5)
   bullseyeWindow -$0000D2(A5)
   qd              -$0000CE(A5)file="MacTraps"
   randSeed        -$000082(A5)
   screenBits      -$00007E(A5)
   arrow           -$000070(A5)
   dkGray          -$00002C(A5)
   ltGray          -$000024(A5)
   gray            -$00001C(A5)
   black           -$000014(A5)
   white           -$00000C(A5)
   thePort         -$000004(A5)
```

Here is how to read it:

- The top line gives the segment's name and size. This segment is named `%GlobalData` and is `0x00084E` bytes in size.
- Each of the other lines lists a variable and its address as a negative offset from `A5`. The variable `appleMenu` is at `0x00084E` offset from `A5`.
- At the far right is the name of the file that defines the variable. The variables `appleMenu`, `fileMenu`, `editMenu`, and `widthMenu` are in the file `bullMenus.cp`. The variables `windowBounds` and `circleStart` are in `bullWindow.cp`.

And this is an excerpt from a code segment:

```
Segment "Seg2"    size=$000524rsrcid=2
   SetUpMenus(void)     $000004 file="bullMenus.cp"
   AdjustMenus(void)    $000092
   HandleMenu(long)     $00019A
   InitMacintosh(void)  $00029C file="bullseye.cp"
   HandleMouseDown(EventRecord *)$0002BE
   HandleEvent(void)    $00039C
   main                 $00045E  JT=$000072(A5)
   SetUpWindow(void)    $000472 file="bullWindow.cp"
   DrawBullseye(short)  $0004AA
```

Here is how to read it:

- The top line gives the segment's name, size, and resource ID. The segment `Seg2` is `0x0004FC` bytes in size and is in code resource `2`.
- Each of the other lines lists a function name and its offset within the segment. Notice that when using C++, the compiler adds the type of the arguments to the name for each function. This is done throughout C++ so that you can distinguish between overloaded functions with the same name. These parameter types show up in the debugger displays, MacsBug names, and other compiler output. The function `SetUpMenus(void)` is at offset `0x000004` from the beginning of the `"Seg2"` code segment.
- If a function has a jump table entry, the address of the entry is listed as an offset from `A5`. The function `main()` has a jump table entry at `0x000072` offset from `A5`.
- At the far right is the name of the file that defines the function. The functions `SetUpMenus()`, `AdjustMenus()`, and `HandleMenu()` are in the file `bullMenus.cp`. The functions `InitMacintosh()`, `HandleMouseDown()`, `HandleEvent()`, and `main()` are in `bullWindow.cp`.

pascal keyword

The identifier `pascal` is reserved to define functions that follow Pascal calling conventions.

4 Using the Symantec C++ Compiler

Symantec C++ THINK Inspector 5

Inspectors work in a manner fundamentally different from that of debuggers. A debugger lets you examine the state of a program from the point of view of variables that are explicitly in the source text of a program. An inspector helps to uncover objects present in the program's address space wherever they occur, even if no variables reference them explicitly.

This chapter describes the operation of the Symantec THINK Inspector.

Contents

5 Symantec C++ THINK Inspector

Quick Start

To use the THINK Inspector, the following must be true about your program:

- It is written in Symantec C++.
- It uses Object Pascal objects or C++ pointer-based objects with virtual destructors.
- It is compiled for debugging and with MacsBug symbols.

To use the inspector, choose **Use Debugger** and then **Run** from the **Project** menu. After the Debugger has finished starting up, choose **Launch Inspector** from the **Inspect** menu. The inspector starts up and presents a hierarchical list of classes.

Features

The inspector provides the following features:

- You can view, hierarchically or alphabetically, your program's classes.
- You can find active objects in memory.
- You can list selected classes' methods and currently existing instances.
- You can direct the debugger's source display to a particular method by double-clicking the method name in the lower pane; option-double-click brings the debugger to the front.
- You can determine the line of code where an object was allocated.
- You can display objects hierarchically to show the nesting of object types.
- You can resize the three panes, and they will maintain a proportional relationship regardless of the window size.
- You can specify the font and size of the text in the panes differently for each pane.

- You can open multiple inspectors simultaneously and store the sizes of the windows from session to session.

Menus

This section describes each of the menus.

File menu

- **New inspector** – creates a new inspector window.
- **Close inspector** – closes the frontmost inspector window (no saving).
- **Page setup**, **print**, **quit** – standard operations.

Edit menu

- Standard editing functions

Classes menu

- **Hierarchical**, **alphabetical** – classes are displayed according to which of these is checked.
- **Expand/collapse subclasses** – active when in hierarchical display, performs all at once expansion/ contraction of the display at the selected class.
- **Expand all subclasses** – exhaustively expands the hierarchical display.
- **Find next active class** – searches in memory for the next class in the list that has active instances.
- **Find all active classes** – selects classes that have active instances.

Inspect menu

- **Instances** – evaluates the selected objects in the upper-right pane and puts the results into the THINK Debugger's data display already expanded.
- **Function** – moves the THINK Debugger's code window to the beginning of the selected method in the bottom pane.

- **Both** – evaluates selected objects and shows the selected method.
- **Update value** – re-evaluates the selected and expanded object in the upper-right pane.
- **Show allocation** – locates the line of code on which the selected object was created.

Font and size menus

- Standard font and size operations.

Inspector Window

The following figure shows a typical inspector window:

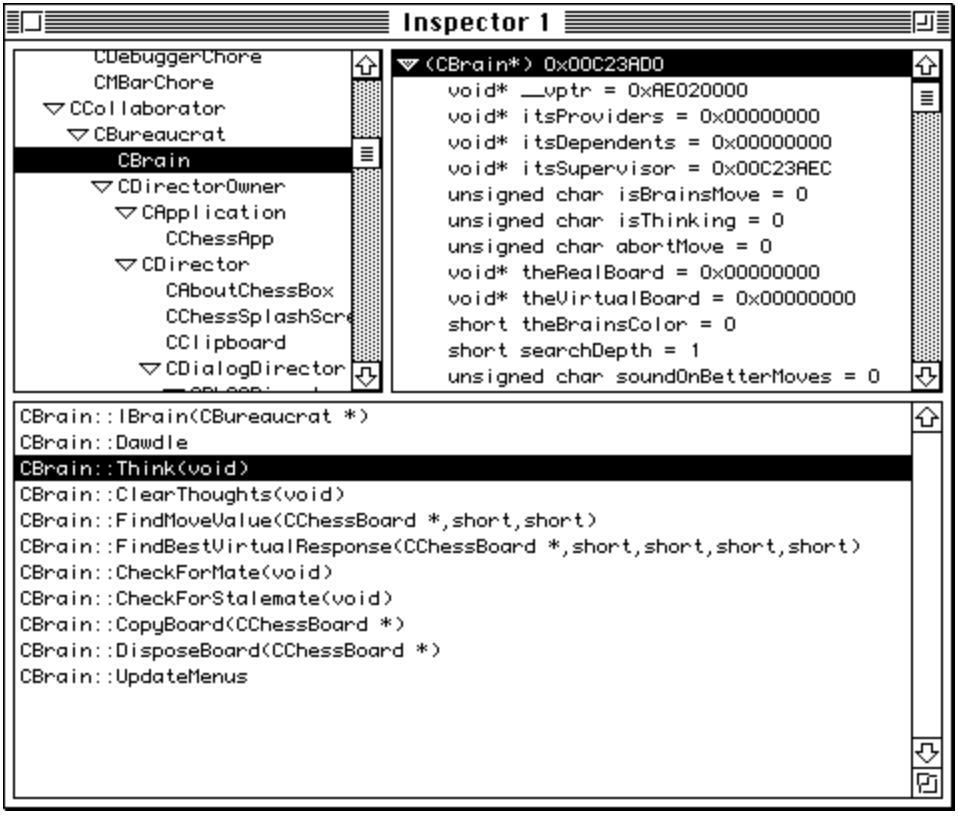


Figure 5-1 Inspector window

This inspector is in hierarchical display mode and has one class selected: `CBrain`. There is a single instance of `CBrain`, which is shown in the upper-right pane, fully expanded. The member function `CBrain::Think` is selected in the lower pane.

The size of the panes can be altered by clicking the mouse between the panes. If you click the space between the upper panes, the widths of the upper panes are adjusted. If the mouse is clicked between the bottom and upper panes, the heights of all panes are adjusted. Finally, if you click the mouse in the center point where all panes meet, both dimensions are adjusted. Three different cursors are used to represent these adjusting options, as shown below:

Figure 5-2 Horizontal adjuster

Figure 5-3 Vertical adjuster

Figure 5-4 Both dimensions adjuster

5 Symantec C++ THINK Inspector

Language Reference 6

This chapter describes in detail aspects of the Symantec C++ implementation that are not part of the C++ language definition.

Contents

How Symantec C++ Implements C++

Symantec C++ supports 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.

Identifier length and capitalization

Symantec C++ allows up to 256 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 C++ Looks for Header Files

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

#include statement	**Symantec C++ looks here**
`<filename.h>`	Symantec C++ looks only in the THINK Project Manager tree.
`"filename.h"`	Symantec C++ looks first in the referencing folder, then in the project tree, and finally in the THINK Project Manager tree.

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++ looks 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. You can use the `#pragma once` directive to do this.

If you have the directive:

```
#pragma SC once
```

in your header file, Symantec C++ includes that file only once. If another file tries to include that header file, Symantec C++ knows that the symbols in that file have already been defined, so it doesn't process the file again.

Note

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

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++ ignores 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++ 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++ will search that folder.

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

Using aliases

Symantec C++ lets you work with the alias of a project file. The project tree begins where the original project is. However, Symantec C++ does not support aliases in these cases:

- Putting aliases in a project=
- Including aliases in an #include statement
- Using an alias as a project's resource (.rsrc) file
- Inserting an alias of a folder in your THINK Project Manager tree or project tree, except in a folder called Aliases in the THINK Project Manager tree or project tree

Using the trees

The way Symantec C++ keeps track of your files 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 the Symantec C++ and project trees.

Don't put project folders in the THINK Project Manager tree

This is the most common mistake. It seems natural to put all your Symantec C++ files in one folder, then toss your project folders in there as well. If you set up your disk like this, Symantec C++ searches all your other project trees every time it searches the THINK Project Manager tree. Setting up your project folders this way not only increases search time, it also increases the likelihood of duplicate names within trees.

Avoid duplicate file names in trees

Just as you can't have two files with the same name in the same folder, you shouldn't have duplicate file names in different folders within the project or THINK Project Manager 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 OK to have the same file name in both the project and THINK Project Manager trees. Symantec C++ resolves the conflict by search order.

Using Register Variables

The following table shows how Symantec C++ supports registers:

Register	Defined to be
`D0`, `D1`, `D2`, `A0`, and `A1`	Scratch registers, which are not preserved by functions
`A5`	Global variable and jump table pointer
`A6`	Local frame pointer
`A7`	Stack pointer

Table 6-1 Register assignments

Alignment

In general, variables are aligned on 16-bit word boundaries.

By default, structure members align on word boundaries. The exceptions to this rule are:

- Structure or class members that are character arrays that align on byte boundaries
- Where two or more single-character variables follow each other, the first character is aligned on a word boundary, whereas subsequent ones are byte-aligned

This structure alignment is useful for defining a structure that maps onto a hardware device or a predefined data element. This alignment control is only valid within structures; everything else is still aligned on word boundaries.

Warning

You must compile each source file referencing a structure with the same type of alignment. If two of your files are compiled with different alignments but reference the same structure, the resulting error messages flag a condition that is very hard to track down when debugging.

The _new_handler

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 shown below.

```
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.

Internal limits

The following table specifies how big you can make certain aspects of your code.

Description	Limit
Characters in a line	No limit
Characters in an identifier	256
Characters in an external identifier	256
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 `#include`s that can be nested	No limit
Number of `#include` paths	No limit
Number of `#if`s that can be nested	No limit
Number of command line arguments	No limit

Table 6-2 Internal limits of Symantec C++ implementation

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.

Integer Representation

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

Floating-Point Representation

You 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*.

The Apple Numerics Manual, Second Edition *(Addison-Wesley) by Apple Computer documents those formats in detail.*

Symantec C++ uses four different representations for floating-point values:

- 4-byte IEEE single precision
- 8-byte IEEE double precision
- 10-byte SANE extended precision
- 12-byte MC68881 extended precision

A `float` uses the 4-byte IEEE representation. The representations for `double` and `long double` depend on the option settings, as shown in the following table:

If 8-byte doubles is...	and Generate 68881 is...	Then doubles format is...
On	On or Off	8-byte IEEE
Off	Off	SANE extended
Off	On	MC68881 extended

Table 6-3 Option settings and resulting doubles format

If you're writing a library for general use, or if your project contains any libraries or projects compiled with the SANE extended format, use the SANE extended format. These Symantec C++ libraries use the SANE extended format:

- ANSI++
- profile++
- unix++

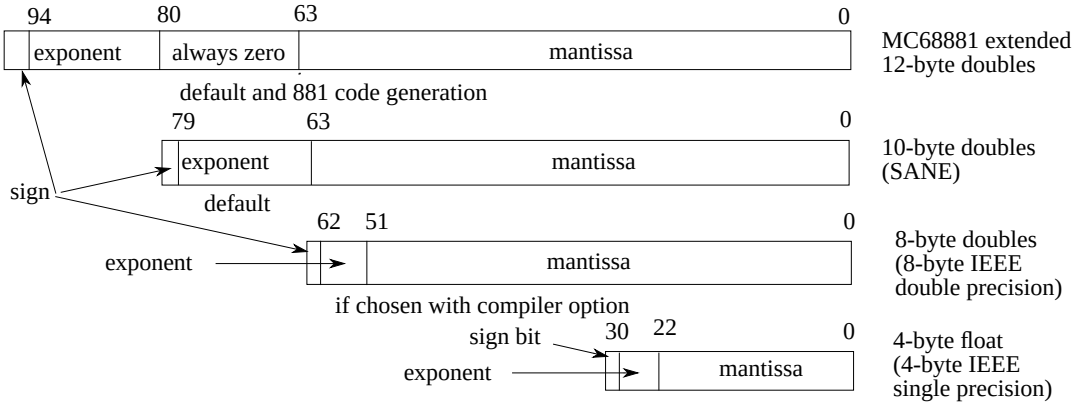


Figure 6-1 Floating-point representation

Removing Symantec C++ Extensions

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

Strict ANSI conformance

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

- These keywords are not recognized:

```
asm                 _cdecl
cdecl               __fortran
__handle            _inf
```

```
inherited          __machdl
__nans             __nan
pascal             _pascal
__pasobj
```

- Predefined macros that do not start with a single or double underscore are not defined.
- You cannot use arithmetic on pointers to functions.
- 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 `'???\?'`.
- 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*/`.
- Empty member lists in `enum` declarations and member lists with a trailing comma are syntax errors.
- You cannot use binary numbers such as `0b10110`.
- At least one hexadecimal number must follow a `\x` escape sequence.
- The program must end with a newline. Each translation unit must end with a newline.
- Enums may be 1 byte, 2 bytes, or 4 bytes long. If the enums are always ints option of the Language Settings page is on, enums are always 4 bytes long.
- You cannot obtain the size of a function with `sizeof`.
- You cannot use the operators `!<`, `<>`, and `!>`.
- You cannot use hexadecimal floating-point constants.
- You cannot use `#ident`.

- You cannot cast an `lvalue` to a different type.
- Anonymous unions must be static.
- A noninteger expression is not converted to an integer expression where a constant expression is required.
- 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;
}
```

- A reference cannot be generated to a temporary.
- The type `void *` is not compatible with other pointer types.
- You cannot convert to and from a `void`.
- You cannot type something `void` where a value is required.
- You cannot put a `sizeof` or a `cast` in a preprocessor expression.
- You cannot use the pre-increment or post-increment operator function as an overloaded function for post increment or post decrement.

Relaxed ANSI conformance

The Relaxed ANSI conformance option on the Language Settings page includes all items in the above list except numbers 1 and 12. These two items are of specific relevance to Macintosh programming. Use this option to ensure strict type checking while retaining the extensions necessary for native Macintosh programming.

Predefined Macros

Symantec C++ predefines these macros:

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

__SC__, THINK_CPLUS
The hex version number of Symantec C++. The current version is 0x700. `THINK_CPLUS` is defined only for the integrated Symantec C++ translator, to distinguish it from the Symantec C++ for MPW translator. All Symantec C/C++ compilers define `__SC__`.

macintosh, MC68000, mc68000, m68k
One.

mc68881
One, if the option Generate 68881 instructions is on.

__cplusplus
One.

__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.

__TIME__
Set to a string containing the current time.

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

__FAR_CODE__
One, if the project type is set to “Far CODE.”

__FAR_DATA__
One, if the project type is set to “Far DATA.”

__A4_GLOBALS__
One, if the project type is a non-application.

Note that __STDC__ is never defined (it indicates ANSI C conformance) and that applec is never defined (Symantec C++ for MPW defines it to be one).

#pragma Directives

Symantec C++ implements the following #pragma directives:

```
align           once
message         parameter
noreturn        template
trace on        template_access
trace off
```

The Symantec C++ compiler ignores the segment pragma. 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++ compilers. If you do not specify SC and the pragma directive is not one of the nine listed above, then the Symantec 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 2, which maximizes performance on systems with a 16-bit bus. If you use this pragma with no argument, the compiler uses the default setting.

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 `#include`-ing 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 depend on it.

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 private
```

Public access means that templates are expanded as usual and that their names are globally accessible. If one template is expanded with public access in two different files, you get an 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++.

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. Capitalization is significant only in inclusion of header files. For example,

```
#include "one.h"
#include "ONE.h"
```

includes `one.h` two times even though it contains a `#pragma sc once.`

pragma SC parameter

This directive applies to a subsequent inline function definition and allows parameters to be passed in registers instead of on the stack. It specifies which register holds the return value and which registers parameters are passed in. The `#pragma parameter` directive must appear before the inline declaration.

`#pragma SC parameter` *return-reg*$_{opt}$ *function-name (param-reglist*$_{opt}$*)*

Function-name is the name of a function that is subsequently defined inline. If the definition is not defined inline, never defined, or already defined, the directive is ignored.

The optional *return-reg* can be `__A0`, `__A1`, `__D0`, or `__D1`.

The optional *param-reglist* is a parameter list made up of `__A0`, `__A1`, `__D0`, `__D1`, or `__D2`.

The inline definition must have a prototype. The return type must be an integer type or a pointer type. The return type must be 4 bytes long if the return is in an address register. The argument types may not exceed 4 bytes, except when the inline definition is declared `pascal`, in which case the address of the parameter is used. An address register can hold a 2-byte or 4-byte value for arguments, but in no case can an address register hold a 1-byte value.

Examples:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size
byteCount)
 = 0xA322;

#pragma parameter Delay(__A0,__A1)
pascal void Delay(long numTicks,long
*finalTicks)
 = {0xA03B,0x2280};
```

#pragma SC message

This pragma causes the compiler to print the specified text 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 trace on

Inserting `#pragma SC trace on` in the Prefix generates calls to code profiler routines that collect timing statistics about your function. The syntax is:

```
#pragma SC trace [on]
```

Using `#pragma SC trace on` has the same effect as selecting the Generate profiler calls option in the **Symantec C++ options** dialog box. (See Chapter 7, "Compiler Options Reference.")

Symantec C++ can profile only functions that have stack frames. To create stack frames for most functions, turn on the Always generate stack frames option in the **Symantec C++ options** dialog box and, for inlines, the Use function calls for inlines option.

#pragma SC trace off

Inserting `#pragma SC trace off` in the Prefix turns off calls to code profiler routines. The syntax is:

```
#pragma SC trace [off]
```

Using `#pragma SC trace off` has the same effect as deselecting the Generate profiler calls option in the **Symantec C++ options** dialog box.

Using Pascal Object Classes

THINK C, MPW Pascal, THINK Pascal, and Symantec C++ have some object-oriented extensions that can define Pascal records similar to C++ classes. The keyword `__pasobj` defines a C++ class that is compatible with a Pascal object. If `__pasobj` is placed after the keyword `struct` or `class`, the class is in relocatable memory in the form of a Pascal object. The objects are accessed via Macintosh handles.

`PascalObject` is a single inheritance model supporting inherited `member()` and `new_by_name()`. Data for class objects is also allocated using `NewHandle`. To create an object hierarchy based on `PascalObject`, use either the name `PascalObject` as the base derivation or use the special name `__pasobj`. For example:

```
class TObject : PascalObject {
//...
}
```

or

```
class __pasobj TObject {
//...
}
```

Note

To use PascalObject, you need to #include <Types.h>.

The name __pasobj must be used when forward-declaring a class implemented as PascalObject:

```
class __pasobj TShape;
class __pasobj TOval;
```

This is necessary because the mangled name for a pointer to a PascalObject differs from a pointer to a non-PascalObject.

Pascal object extensions to Symantec C++

The following extensions to Symantec C++ exist only when you are using PascalObjects. Be sure that you #include <oops.h> in the source file that uses these functions.

```
void *new_by_name(char *aName);
```

Creates a new object. aName is a C string that names the class. If there is no class with the given name, new_by_name returns NULL. The named class must be defined in your program. If your program defines a class but never creates an instance of it, the smart linker may remove the class definition. In this case, you'll be unable to use new_by_name to create an instance of that class. One way to get around this restriction is to create a dummy object, then immediately delete it. Another way is to use the member() function to refer to the class.

```
char *class_name(void *anObject);
```

Returns a string that is the name of the class to which the anObject belongs. This function does not check anObject to make sure that it is a valid object reference.

```
char member (void *anObject, void *aClass);
```

Returns 1 if anObject is an instance of aClass or an instance of one of its ancestors; zero otherwise.

```
void set_class_index pascal void(
*methtable(), void *obj);
```

Sets the class index for a `PascalObject`. You need to use this function only if you make a custom allocator for your `PascalObject` class. It sets the class ID so that virtual functions are called correctly.

Pascal handle-based classes have the following restrictions, in addition to the restrictions noted above for all handle-based classes:

- You cannot declare nonvirtual member functions in Pascal code or call them from Pascal code. You can declare member functions as either Pascal or non-Pascal, but you can call non-Pascal member functions only from C++ code.
- You can use constructors and destructors, but they can't take any arguments. If a class has virtual member functions, then the destructor should also be virtual. Since you can write a conventional cleanup routine to call from Pascal, you can call it in the destructor. Don't call the Dispose method of your object from within your destructor, as it will try to delete the object a second time.
- Overloading, type conversion, and operator functions are not allowed for virtual members of Pascal classes or for any function with the `pascal` attribute. These features require type signatures, which the Pascal naming conventions do not support.
- Overloading, type conversions and operator functions are allowed for member functions of Pascal classes, but they cannot be declared or accessed from Object Pascal.
- You cannot cast a pointer to a Pascal handle-based class to a pointer to a non-Pascal handle-based class, and vice versa.
- 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 (*) ()`.

- You cannot declare global variables, local variables, arrays, members, or parameters of handle-based classes (rather than pointers to them).

Using the Macintosh Handle Pointer Type

The Macintosh handle pointer type implements a compiler-supported scheme for using Macintosh relocatable memory in a portable manner. It is a compatible extension to ANSI C and is compatible with C and C++.

A Macintosh handle is a type used to refer to the data rather than a normal pointer to pointer. To refer to the data, the handle must be accessed, in other words, converted into a pointer by a dereference. Traditionally, the user was required to dereference, which produced code not portable to other environments. The Symantec C++ implementation removes the requirement to dereference the handle from the developer and places the responsibility with the compiler.

The __machdl pointer

This additional pointer type is attached to a '*.' Declarations look like:

```
long __machdl *h;
   /* h is a handle to a long */
```

A more typical Macintosh declaration would be:

```
long **h;
   /* h is a handle to a long */
```

Macintosh handles modify the declaration from <*pointer to*> to <*mac handle pointer to*>. `__machdl` is right-associative. A Macintosh handle pointer is a 32-bit type and follows the same rules as pointers.

Dereferencing a handle

Conversions from handles to pointers occur whenever a handle pointer is dereferenced, or when a handle is cast to a pointer. The conversion is done by the compiler by generating a pointer dereference in the code. For example, the following operations all convert handles to pointers.

```
int __machdl *h;
struct A __machdl *h2;
int *f;
int i;
extern void func(int *pi);

f = h;
*h = i;
h[3] = *f;
i = *((int *)h + 6);
h2->b = i;
func(h);
```

The same code done explicitly by the user would be:

```
int **h;
struct A **h2;
int *f;
int i;
extern void func(int *pi);

f = *h;
**h = i;
(*h)[3] = *f;
i = *(*h + 6);
(*h2)->b = i;
func(*h);
```

The conversion is done for the handle every time the compiler decides that the previous conversion is invalid.

The optimizer is aware of handles as a special type and determines when a new handle dereference is necessary or when a previous one can be used instead. Consider:

```
struct { int a,b; } __machdl *h;
h->a = 1;
h->b = 2;
```

To convert h to a pointer once, the optimizer changes the code to:

```
struct { int a,b; } __machdl *h, far *p;
p = h;
p->a = 1;
p->b = 2;
```

The result of a previous conversion cannot be used if:

- The value of the handle might have changed.
- A handle dereference was carried out on another handle.
- A function was called (because that function may convert other handles, resulting in the previous case).

Handle dereferencing is slower than normal pointer access. To speed up operations, you can convert a handle to a pointer yourself; if you know that a called function does not move memory, your conversion is still valid after the function call is made. (See *Inside Macintosh* for a description of when the operating system will move memory.)

Storage allocation

Storage allocation is done by using the NewHandle toolbox routine.

Portability

Using Macintosh handles allows the __machdl keyword to be defined and the storage allocation routines to be redefined to standard memory allocation routines. For example:

```
#define __machdl
#define NewHandle malloc
```

The code will still port to a flat memory environment. The code explicitly dereferencing handles cannot be ported to a flat space.

Placing C++ classes in handle memory

The __machdl keyword may also be used to indicate a class storage type. If __machdl follows the keyword struct or class, the class is in relocatable memory. All derivations of the class are allocated in relocatable block memory, and all class pointers are implicitly Macintosh handle pointers.

For example:

```
class __machdl RelocatableObject {
public:
   int x;
// other fields
}

RelocatableObject *object;
   /* this is an implicit mac handle */
```

The default `new` and `delete` operators are implemented via the Macintosh Toolbox Memory Manager handle routines. As with explicitly declared Macintosh handle pointers, the compiler generates the required double dereference.

For example, to access field x:

```
object->x
```

Handle-based classes have the following restrictions:

- You cannot declare global variables, local variables, arrays, members, or parameters of handle-based classes (rather than pointers to them).
- You cannot use multiple inheritance with handle-based classes.
- You can create handle-based objects only with the `new` operator. The only use of a dereferenced handle-based class pointer (for example, `*x`) is to refer to a field in the class (for example, `*x.y` or `x->y`).
- You cannot cast pointers to handle-based classes to any other type except to a pointer to another handle-based class. You cannot cast pointers to anything else to a pointer to a handle-based class.
- You cannot perform address arithmetic on a pointer to a handle-based class, except the implicit arithmetic used in a member reference.
- Avoid taking the address of a field belonging to an indirect class (for example `&x->y`). It is legal, but unsafe, because the object may move. This restriction includes

the implicit use of pointers by references, such as `int& p = x->y`.

- You cannot allocate an array of handle-based objects; for example, `new MyObjects[10]`.

Debugging programs that use handles

Debugging C++ applications that use a lot of dynamically allocated memory is notoriously difficult. Experienced C++ programmers have learned to live with this.

Known danger zones include:

- Function calls that could dereference handles or move memory.
- Converting code from using `malloc` to `handle_malloc`. When doing so, watch out for:

  ```
  p = (char *) malloc(n);
  ```

  to

  ```
  h = (char *) handle_malloc(n);
  ```

  instead of the correct:

  ```
  h=(char __handle *) handle_malloc(n);
  ```

The cast to `(char *)` dereferenced the handle and stored a pointer into `h` instead of the required handle.

To deal with these problems:

- While writing and debugging the program, disable handles with `NO_HANDLE`. If handles are then enabled and the program fails, you can confine your search to the handle pointers.
- If possible, encapsulate data structures that use handles into C++ classes. This confines the code that dereferences the handles to a few places and isolates the problems within a class definition.
- Program defensively. Assume that all function calls invalidate previous handle dereferences.

The Inherited Keyword

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. It is supported in both Symantec C++ and THINK C with object extensions.

Inline Function Definitions

You can define inline functions using this form:

```
returnType functionname (arguments) =
    { instr1, instr2, … };
```

Symantec C++ replaces a call to the function *functionname* with the machine instructions *instr1, instr2, …* instead of generating a function call. For example, the function `PrOpen()` is defined as follows:

```
pascal void PrOpen(void)
   = {0x2F3C,0xC800,0x0000,0xA8FD};
```

When you call `PrOpen()`, Symantec C++ generates these instructions:

```
move.l   #$C8000000, -(AT)
dc.w     $A8FD
```

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

You can use the `#pragma parameter` directive to assign registers to parameters:

```
#pragma parameter __A0 NewHandleClear(__D0)
pascal Handle NewHandleClear(Size
byteCount)
 = 0xA322;
```

Most of the Macintosh Toolbox routines are defined in the header files as inline functions.

6 Language Reference

Compiler Options Reference 7

This chapter describes the Symantec C++ Options menu commands. Within each menu, commands are described in the order in which they appear.

Contents

7 Compiler Options Reference

The Options Menu

Use the **Options** menu to choose Symantec C++ compiler options. There are six types of options, as shown in Figure 7-1. Each is described on one of six pages 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 reach the Prefix page, you have to press the Command key along with an arrow key.)

Figure 7-1 The Options menu

The six types of compiler options are as follows:

- Language Settings lets you choose extensions to the C++ language.
- Compiler Settings lets you control how Symantec C++ generates code.
- Code Optimization lets you control how Symantec C++ optimizes your code.
- Debugging lets you specify how the Symantec C++ debugger works.
- Warning Messages lets you control which warning messages (if any) Symantec C++ generates.
- Prefix lets you write code that Symantec C++ 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. Use the Copy button at the top of the page to copy the Symantec C++ defaults to the current project, or if you want the options you've set for a particular project to be the Symantec C++ options. Note that

even though only one page of the options shows up on the screen at a time, the Copy button copies the settings for *all* of the options at once.

When you click the Factory Settings button at the bottom of the page, Symantec C++ sets the options on all pages back to their original settings. These original settings are noted in this chapter.

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

Language settings

On the Languages Settings page, you can choose whether the Symantec C++ compiler uses extensions to the C++ language.

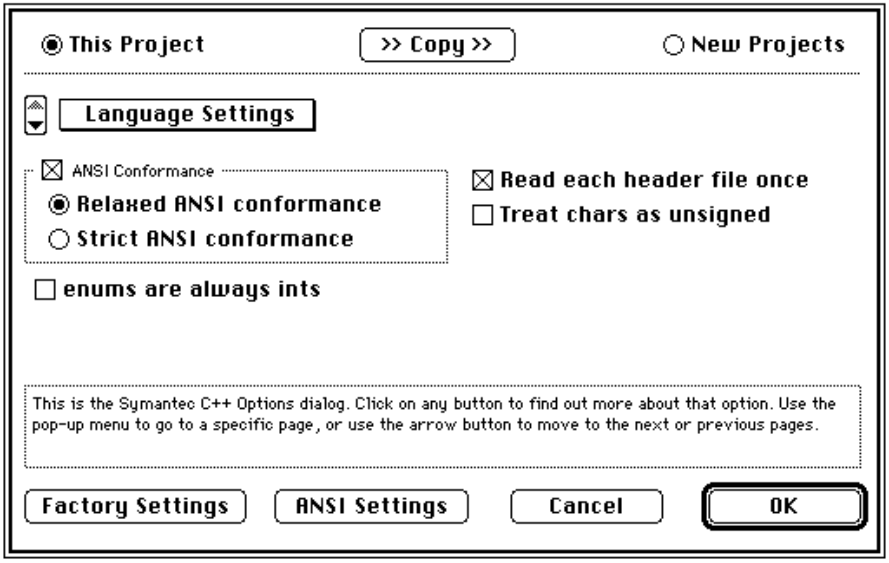


Figure 7-2 The Language Settings page

Use the options on this page to decide how closely Symantec C++ conforms to the ANSI draft description of the C++ language. This is the only page 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.

The ANSI conformance options are described in detail in the sections "Strict ANSI conformance" and "Relaxed ANSI conformance" in Chapter 6.

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

Relaxed ANSI conformance	This option is the same as 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.

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`, `int`, or `long 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 C++ makes enumeration constants as small as possible. And, if necessary, it makes a constant as large as a `long 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 a `long int`:

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

If this option is on, you cannot use constants as large as a `long int`.

Read each header file once
If this option is on, Symantec 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.

Treat chars as unsigned
If this option is on, Symantec 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.

Compiler settings
The Compiler Settings page lets you control how Symantec C++ compiles your code.

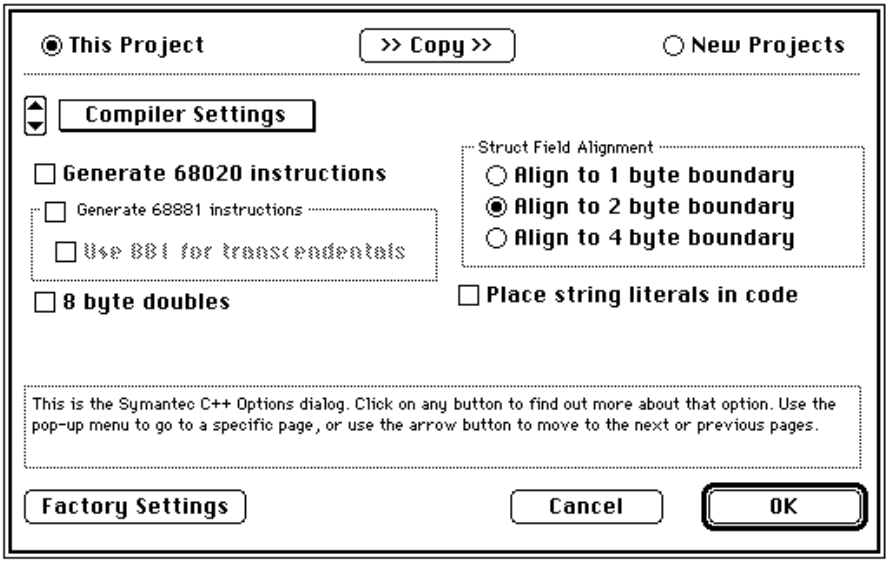


Figure 7-3 The Compiler Settings page

Generate 68020 instructions
If this option is on, Symantec C++ generates code that is optimized for Macintoshes with the MC68020, MC68030, or MC68040 (such as the Macintosh LC, II, IIfx, or Quadra) and that does not run on Macintoshes with the MC68000 (such as the Macintosh Classic or Plus). If the option is off, Symantec C++ generates code that runs on all Macintoshes. The original setting is off.

If the option is on, Symantec C++ generates MC68020 instructions for addressing and long-word multiplication, division, and modulo operations.

Note

Before your program uses MC68020 instructions, make sure you have a MC68020, MC68030, or MC68040 in your machine or your program may crash. Use the Gestalt Manager, as described in *Inside Macintosh.*

Generate 68881 instructions

If this option is on, Symantec C++ generates code that is optimized for Macintoshes with the MC68881 or MC68882 floating-point unit (such as the Macintosh II or IIfx) or the MC68040. The code will not run on Macintoshes without the floating-point unit (such as the Macintosh Classic, LC, or Plus). If the option is off, Symantec C++ generates code that runs on all Macintoshes. The original setting is off.

If this option is on, Symantec C++ generates code for the floating-point coprocessor.

Use 881 for transcendentals

If this option is on, the math coprocessor is used for transcendental math functions such as sine or cosine. The original setting is off.

Note

Before your program uses MC68881 instructions, make sure you have a MC68881 or MC68882 in your machine or your program may crash. Use the Gestalt Manager, as described in *Inside Macintosh.*

8-byte doubles

When this option is on, doubles are the same size as short doubles, which are 8-bytes long. When it's off, they're the same size as long doubles. The original setting is off.

Use the 8-byte doubles option if you're porting code from a compiler that uses 8-byte doubles, such as MPW C. Otherwise, leave the option off and Symantec C++ makes doubles and long doubles the same size.

Struct field alignment

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. This option, which is the most common form of structure alignment, is the default. The original setting is on.
Align to 4-byte boundary	This option pads out all fields in structures, unions, and classes to 4-byte (long-word) boundaries. For machines with MC68020 or higher processors, this option can increase speed. The original setting is off.

Place string literals in code
With this option on, string literals are placed in code segments instead of global data space. String literals are accessed using PC relative addressing rather than A5 relative addressing. The original setting is off.

Note

This rule applies only to string literals inside functions. For example:

```
void f () {
 char * x=”one” //“one” goes into code
 }
char * x=”two”; //“two” goes into data
```

Code optimization

The Code Optimization page lets you control how Symantec C++ optimizes your code.

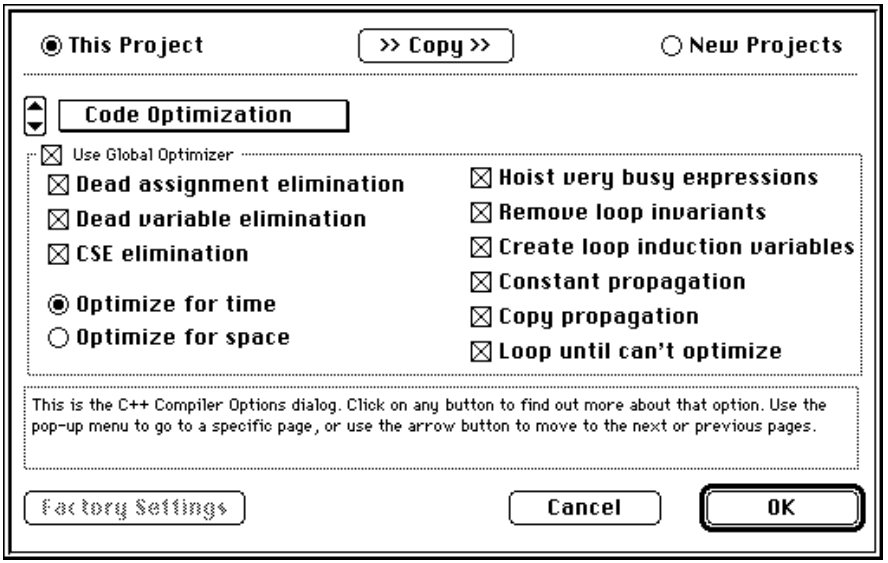


Figure 7-4 The Code Optimization page

Use global optimizer

This option controls the global optimizer. It is a "master switch." If it's off, the options below it are disabled. If it's on, those options are enabled, set to the same values as the last time the master switch was on. 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 significantly different from your source code. The debugger may not be able to pick out the machine code instructions that correspond to a given statement.

Dead assignment elimination

If this option is on, Symantec C++ removes assignments to variables that are not used after being assigned, making your code smaller and faster to execute. This optimization also allows Symantec C++ to reuse registers for more than one variable. The original setting is on.

If the option is on, Symantec C++ doesn't 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++ loads `j` from memory into a register twice, once for each time it appears. If this option is on, Symantec C++ loads it from memory only once.

If you're debugging, you may want to turn the option off. When you set a variable in a data window, the debugger puts the new value into memory. If the option is on, however, your program may 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
With this option on, Symantec C++ determines the live ranges of variables in your code, removing any variables that have empty live ranges. The original setting is on.

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

CSE elimination
This optimization makes your code smaller and faster. The original setting is on. Selecting CSE 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 like this:

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

Use this optimization on all your code.

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

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

Hoist very busy expressions
With this option on, the compiler produces a single version of an expression that occurs over several different paths in the code. The result is smaller code. The original setting is on.

Remove loop invariants
This optimization makes your loops faster. The original setting is on. Selecting this option 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
Selecting 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();
```

With the option off, the compiler performs a multiplication each time it figures the address for the next array element (`i * sizeof(int)`). With the option on, the compiler remembers the address of the last element and adds the size of an element to that address. Use this optimization if your code has a lot of loops. Note, however, that using it may make your code slightly larger. The original setting is on.

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)
```

The original setting is on.

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, that can be removed.

The original setting is on.

Loop until can't optimize
If this option is on, Symantec C++ repeatedly runs all selected optimizations until it discovers no further opportunities for optimization. The original setting is on.

Note

Some optimizations are performed sequentially. Occasionally, one optimization creates the opportunity for other optimizations. If this option is on, each optimization is run repeatedly until your code is optimized maximally.

Debugging

The Debugging page lets you specify how the Symantec C++ compiler generates code for debugging.

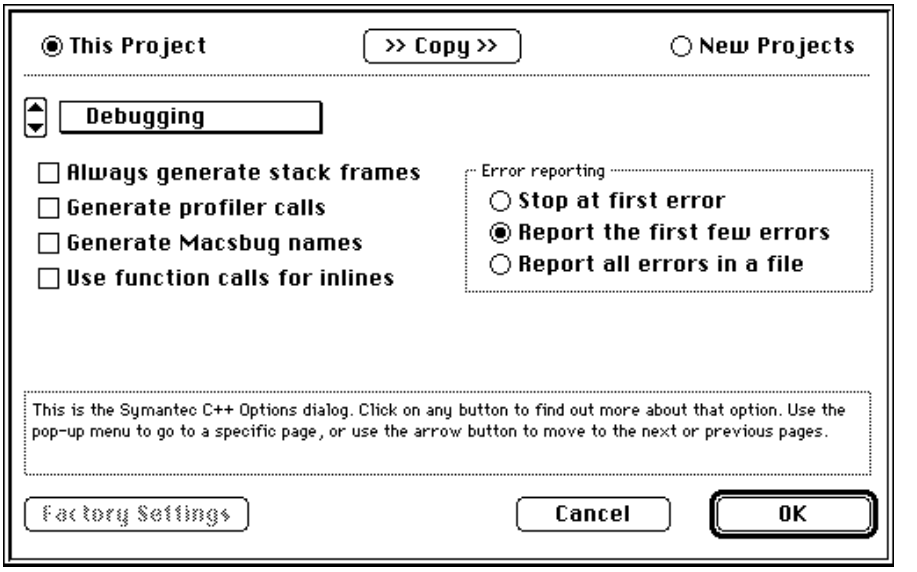


Figure 7-5 The Debugging page

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

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

Generate profiler calls
When this option is on, Symantec C++ generates calls to code profiler routines. The code profiler collects timing statistics about your functions. The original setting is off.

Symantec C++ can profile only functions that have stack frames. To create stack frames for most functions, select the Always generate stack frames option. For inlines, select the Use function calls for inlines option, described below.

Generate MacsBug names
With this option on, Symantec C++ includes function names in your compiled code for assembly language level debuggers such as MacsBug or TMON. The original setting is off.

Symantec C++ generates symbols only for functions that have stack frames. To create stack frames for almost every function, turn on the Always generate stack frames option. Be aware that while MacsBug symbols are useful for debugging, they add at least 8 bytes to every procedure.

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

Error reporting
Each of these three options produces a different error report:

Stop at the first error	If this option is on, Symantec C++ stops 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++ reports the first few errors in your code or stops at the first unrecoverable error. The original setting is on.
Report all errors in a file	If this option is on, Symantec C++ reports errors it finds in your source file or stops at the first unrecoverable error. The original setting is off.

Warning messages

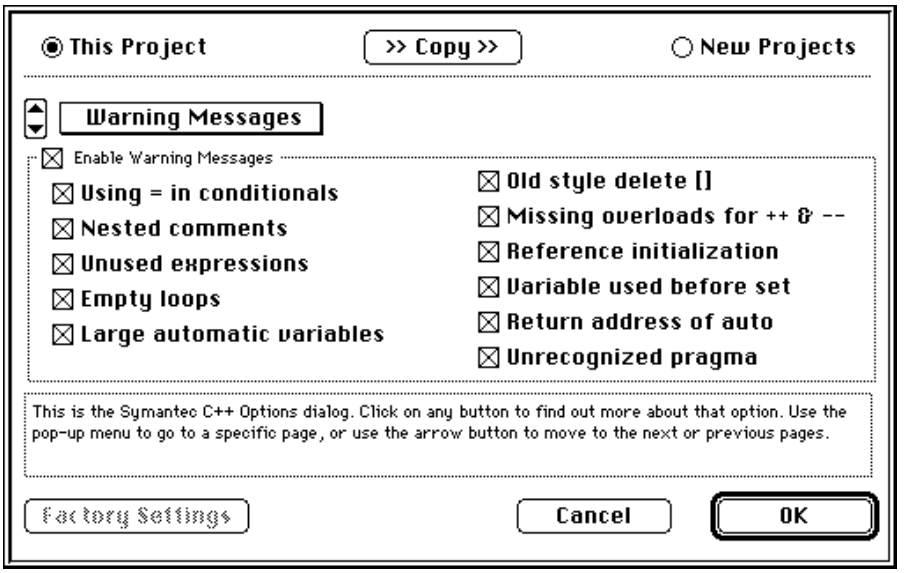


Figure 7-6 Warning Messages page

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

Choosing Factory Settings causes all 11 warning message options to revert to their original settings. By default, the Enable Warning Messages option is set, which automatically sets all warning message options.

The next 11 sections describe these messages and the conditions under which the compiler generates them.

Using = in conditionals
If this option is on, Symantec C++ warns 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) { . . . }
```

Nested comments
If this option is on, Symantec C++ produces 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
```

Unused expressions
If this option is on, Symantec C++ alerts 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
```

Empty loops
If you enable this option, Symantec C++ produces 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;
```

Large automatic variables
With this option, Symantec C++ warns 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` may be preferred. The original setting is on.

Old style delete [ ]
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
```

Missing overloads for ++ & --
This option produces a warning when you have used 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)
```

Reference initialization
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
```

Variable used before set
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
```

Return address of auto
If you enable this option, Symantec C++ produces 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'
}
```

Unrecognized pragma
With this option, Symantec C++ produces a warning when it does 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
```

Prefix

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

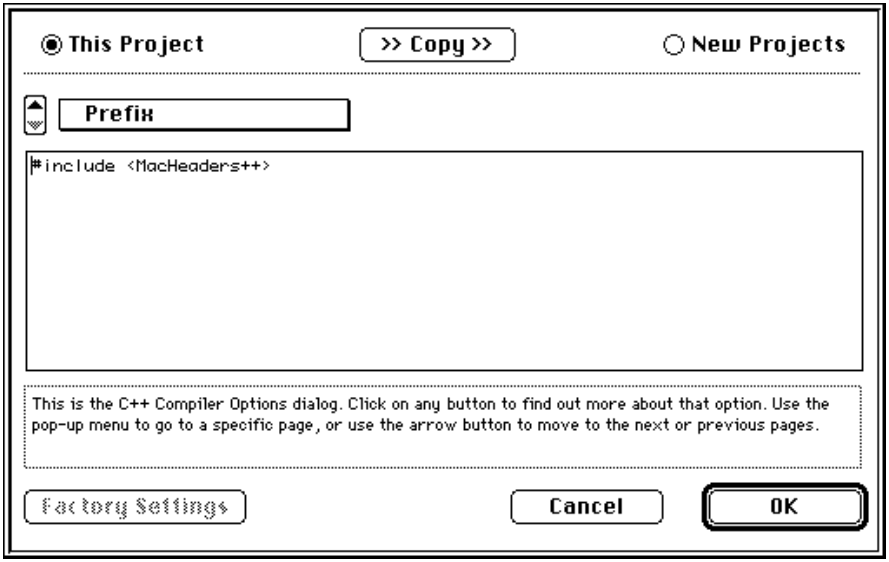


Figure 7-7 The Prefix page

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

If you use a precompiled header file, such as `MacHeaders++`, include it here. By default, this page contains the line: `#include <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++ source file in your project.

Porting Code A

This chapter helps you port existing code to Symantec C++.

Contents

A Porting Code

Porting from MPW C++

Wherever possible, Symantec C++ has striven for compatibility with MPW C++. The Symantec C++ calling conventions are the same as MPW C++ except that MPW C++ promotes `char` and `short` arguments to `int`, and MPW C++ promotes `float` and `double` arguments to `long double`. Symantec C++ calling conventions for procedures with a variable number of arguments are compatible with calling conventions in MPW C++.

Symantec C++ mangled names for functions are different from those used in MPW C++. You must recompile source code with the Symantec compiler due to this difference. 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 directory for the file. They do not search the user directories.

Structures as arguments

Symantec C++ passes structures on the stack, while MPW Pascal passes a pointer to the structure and then copies the data into a temporary location. Symantec C++ routines that you declare as `pascal` use the MPW Pascal convention, but they do *not* use a copy constructor to copy the passed structure to the temporary location.

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 of Symantec C++ handle `enum` prototyping differently.

Function prototypes and varargs functions

In the Symantec C++ compiler, a function that you prototype and define using the convention

```
func(type arg, type arg)
```

passes the specified type size. The MPW C++ compiler converts such an argument to a larger type—in other words, `short` to `int`, `enum` to `int`, `float` to `double`. Prototypes for functions compiled with

one compiler and called with the other compiler must use `int` instead of `enum`, `short`, and `char`; these functions must also use `long double` instead of `float` or `double`.

Pascal and handle objects

In the Symantec C++ compiler, you define Pascal objects and handle objects using the additional type modifiers `__machdl` or `__pasobj`. The compiler includes these type modifiers when prototype checking. As in MPW C++, these types are inherited from base classes. The classes `PascalObject` and `HandleObject` are defined in `Types.h` using the extended type modifiers.

Structure definition

The Symantec and MPW compilers both 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.

Calling C++ Functions

C++ arguments

Functions that take a fixed number of arguments are evaluated from left to right and are pushed on the stack as they are evaluated. A called function cleans the arguments off the stack. For functions that take a variable number of arguments, arguments are pushed from

right to left onto the stack. The caller of the function cleans the arguments off the stack. Data types are placed on the stack as follows:

Type	How it's passed
`char`, `unsigned char`	Uses a 16-bit word on the stack with the passed value in the high-order 8 bits.
`short`, `unsigned short`, 2-byte `structs`	Uses a 16-bit word on the stack.
`int`, `unsigned int`, `long`, `unsigned long`, `pointers`, 4-byte `structs`, `enums`	Uses a 32-bit word on the stack.
`float`, `double`, `long double`, `comp`	Passed as 80-bit values (96-bit for 68881) on the stack.
structures larger than 4 bytes	Copied onto the stack. If a copy constructor is available, this is invoked to create a local version of the struct or class for use as the parameter. If there is no copy constructor, a member-by-member copy of the original struct or class is made on the stack as the parameter.

Table A-1 How C++ arguments are passed

C++ return values

This table describes how C++ functions return their values:

Type	How the value is returned
`char`, `int`, `long`, `unsigned chars`, `unsigned int`, `unsigned long`, pointers, `enums`	Returned in `D0`.
`comp`, `float`, `double`, `long double`	Returned as 80-bit extended values (96-bit for 68881) with the low-order 16 bits of register `D0` containing the sign and exponent, `D1` containing the high-order 32 bits of the significand, and `A0` containing the low-order 32 bits of the significand.
structures	Returned by allocating a temporary variable on the stack and passing a pointer to it as the first function parameter. The called function copies the return structure value into this using the copy constructor, and also returns a pointer to it. This technique, which is re-entrant, is appropriate when the number of parameters in the original function call is fixed. When a function takes a variable number of parameters, the structure to be returned is copied into a static area of memory and a pointer to that area is returned. The latter technique is not re-entrant.

Table A-2 How C++ functions return values

Note

When the Generate 68881 instructions option is on, the compiler returns floating-point results (`float`, `comp`, `double`, `long double`) in `FP0`.

Calling Pascal Functions

Because the Macintosh Toolbox routines expect to be called with Pascal calling conventions, you may need to write functions in C++ that behave as though you wrote them in Pascal.

Pascal callback routines

Some Macintosh Toolbox routines take a pointer to another function as an argument. Those routines then call the function you passed in. The function you provide is called a callback routine. The Toolbox routines expect the callback routines to follow Pascal calling conventions.

To write functions that behave as though they were written in Pascal, the function definition must begin with the `pascal` keyword. Make sure you provide a return type. If your function needs to behave like a Pascal procedure, declare the return type `void`.

For the parameter declarations, follow the same rules as for calling Toolbox functions. Remember that non-`var` parameters are passed by value, not by reference. If the size of a non-`var` parameter is greater than 4 bytes, pass its address but do not modify the parameter in your function.

For example, `ModalDialog()` lets you provide a `filterProc` to handle events in your dialog. This is how *Inside Macintosh* declares `ModalDialog()`:

```
PROCEDURE ModalDialog (filterProc: ProcPtr;
   VAR itemHit: INTEGER);
```

`ModalDialog()` expects the `filterProc` to have this declaration:

```
FUNCTION MyFilter (theDialog: DialogPtr;
   VAR theEvent: EventRecord;
   VAR itemHit: INTEGER) : BOOLEAN;
```

In Symantec C++, the declaration for `MyFilter()` would look like this:

```
pascal Boolean MyFilter(DialogPtr
theDialog,
   EventRecord *theEvent, short *itemHit)
```

Note

References can also be used for `VAR PascalObjects` so `MyFilter` could be rewritten as:

```
Pascal Boolean MyFilter (DialogPtr
theDialog, EventRecord &theEvent,
short &itemHit);
```

The call to `ModalDialog()`, then, looks like this:

```
void F(void)
{
   extern pascal Boolean MyFilter
      (DialogPtr, EventRecord*, short*);
   short theItem;

   ...
   ModalDialog(MyFilter, &theItem);
   ...
}
```

Keeping C++ and Pascal on speaking terms can be tricky, but Symantec C++ tries to make it as painless as possible.

Calling Pascal routines indirectly

Only Macintosh Toolbox routines and functions that are declared `pascal` are called using Pascal conventions. If you have a pointer to a Pascal routine, you can call it indirectly through a pointer just as you would call a routine in C++, but the pointer must be of type pointer-to-pascal-function.

For instance, this type definition defines `PFunc` as a pointer to a Pascal function that takes one `short` argument and returns `short`:

```
typedef pascal short (*PFunc)(short);
```

This function takes a Pascal routine as an argument and calls it indirectly:

```
short DoPascal(PFunc theFn, short i)
{  // theFn is a pointer to a pascal
   // function that returns short

i = (*theFn)(i);// call it indirectly

i = theFn(i);  // this works too, but you
               // might not be able to tell
               // that it was called
               // indirectly
   return (i);
}
```

Now suppose you have a function declared `pascal` like this:

```
pascal short Add1(short v)
{
   return v + 1;
}
```

You could call the `DoPascal` function defined above like this:

```
j = DoPascal(Add1, 10);
```

Pascal arguments

Arguments are pushed from left to right onto the stack as they are evaluated. The function cleans the argument off the stack.

This table describes how Pascal passes arguments for prototyped functions.

Type	How it's passed
`char`, `unsigned char`, 1-byte `enums`	Uses a 16-bit word on the stack with the passed value in the high-order 8 bits.
`short`, `unsigned short`, `enums`	Uses a 16-bit word on the stack.

Table A-3 How prototyped Pascal functions pass arguments

Type	How it's passed
`int`, `long`, `unsigned int`, `unsigned long`, pointers, 4-byte `enums`	Uses a 32-bit word on the stack.
`float`, `double`	Passed on the stack.
`comp`, `long double`	Extended to 80-bit values (96-bit for 68881) and passed on the stack.

Table A-3 How prototyped Pascal functions pass arguments *(Continued)*

This table describes how Pascal passes arguments for unprototyped functions.

Type	How it's passed
`char`, `int`, `long`, `short`, `unsigned char`, `unsigned int`, `unsigned long`, `unsigned short`, pointers, structures <= 4 bytes, `enums`	Uses a 32-bit word on the stack.
`float`, `double`, `comp`, `long double`	Extended to 80-bit values (96-bit for 68881) and passed on the stack.
structures > 4 bytes	Copied onto the stack.

Table A-4 How unprototyped Pascal functions pass arguments

Note

Always prototype a function before calling it. Calling an unprototyped function can have unexpected results.

Pascal return values

Pascal functions return their values as shown below.

Type	How it's returned
`char`, `unsigned char`	2 bytes on stack.
`int`, `long`, `unsigned int`, `unsigned long`, pointers, `enums`, structures <= 4 bytes	4 bytes on stack.
`comp`, `float`, `double`, `long double`	`float` is returned as 4 bytes on the stack. `comp`, `double`, and `long double` are returned by allocating a temporary variable on the stack and passing a pointer to the stack before any parameters.
structures > 4 bytes	Returned by allocating a temporary variable on the stack and passing a pointer to it on the stack before any parameters. The called function copies the return structure into this area.

Table A-5 How Pascal functions return values

Note

The size of the `double` depends on the 8-byte doubles compiler setting. See "8-byte doubles" in Chapter 7 for more information.

ARM Conformance B

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 chapter describes how Symantec C++ implements those aspects of the 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. The Gray Book refers to *The C++ Programming Language, Second Edition* by Bjarne Stroustrup, published by Addison-Wesley, Reading, Massachusetts, 1991.

This appendix is organized by the numbered sections of the ARM, with individual subsections marked with applicable references in both the ARM and the Gray Book. The ANSI committee is using the ARM as the basis for its definition of the C++ language; this chapter, however, includes references to both the ARM and the Gray Book.

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.

B ARM Conformance

Contents

The section numbers in this chapter correspond to the sections in the Annotated C++ Reference Manual (ARM).

Lexical Conventions

The section numbers in this chapter 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 256 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.

String literals are distinct; they do not overlap in memory. 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, the default), or long word (4 byte) 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. The sizes of the integer types are:

Type	Bytes
`char`	1
`short`	2
`int`	4
`long`	4

Table B-1 The size of integer types

The header `limits.h` specifies the largest and smallest values of the integral types. The following table 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`

Table B-2 Limits of integral types

Variable	Value	Definition
CHAR_MAX	UCHAR_MAX SCHAR_MAX	Maximum value of `char`
CHAR_MIN	0 SCHAR_MIN	Minimum value of `char`
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 B-2 Limits of integral types *(Continued)*

You can represent floating-point values four ways. The representation you use depends on the options you choose when you compile.

Type	Default Size in Bytes	Format
`float`	4	IEEE single precision
`double`	8	IEEE double precision
`double`	10	SANE extended precision
`double`	12	M68881/2 extended precision
`long double`	10	SANE extended precision
`long double`	12	M68881/2 extended precision

Table B-3 Floating-point values

You can find these formats documented in detail in the *Apple Numerics Manual, Second Edition* (Addison-Wesley) by Apple Computer and the *MC68881/682 User's Manual* (Motorola).

The header `floating.h` defines the characteristics of the floating types. The following table 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

Table B-4 Floating-point limits

Variable	Value	Definition
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
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

Table B-4 Floating-point limits *(Continued)*

Variable	Value	Definition
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
DBL_EPSILON	2.220446e-16	Minimum positive number *x* such that 1.0+*x* does not equal 1.0
LDBL_DIG	19	Decimal digits of precision
LDBL_MANT_DIG	64	Number of base FLT_RADIX digits in mantissa
LDBL_MAX_10_EXP	4932	Maximum integer *n* such that 10 raised to the *n*th is representable

Table B-4 Floating-point limits *(Continued)*

Variable	Value	Definition
LDBL_MAX_EXP	16384	Maximum integer *n* such that FLT_RADIX raised to the *n*th minus 1 is representable
LDBL_MIN_10_EXP	-4931	Minimum negative integer *n* such that 10 raised to the *n*th is within the range of normalized floating-point numbers
LDBL_MIN_EXP	-16382	Minimum negative integer *n* such that FLT_RADIX raised to the *n*th minus 1 is a normalized floating-point number
LDBL_MAX	1.189731e+4932	Maximum representable floating-point number
LDBL_MIN	1e-4926	Minimum normalized positive floating-point number
LDBL_EPSILON	1.088437e-19	Minimum positive number *x* such that 1.0+*x* does not equal 1.0

Table B-4 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 B-4.

§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 B-5 Division by zero

For further information, consult the *Apple Numerics Manual, 2nd Edition* or the *68881/682 User's 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 constantness 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. 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:

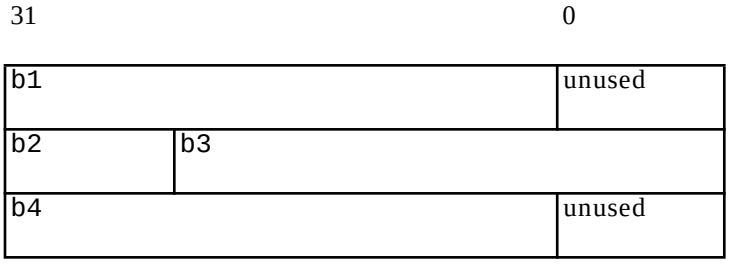


Figure B-1 Sample code as represented in memory

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.

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>`	Looks only in the THINK Project Manager tree
`"filename.h"`	Looks first in the referencing folder, then in the project tree, and finally in the THINK Project Manager 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 nine pragmas. The preprocessor produces a warning for unrecognized pragmas. See “#pragma Directives” in Chapter 6.

§16.10 Predefined Names

[ARM p. 379, Gray p. 614]

Symantec C++ does not define the predefined name `__STDC__` .

B ARM Conformance

Symantec C++ Errors C

This appendix lists and describes error and warning messages generated by Symantec's C++ compiler. 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

Messages marked *Warning* indicate code that does compile but that may not execute as you expect. This appendix lists messages in alphabetical order.

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

C Symantec C++ Errors

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 Compile Errors window, as in the figure below, and continues compiling your project.

Figure C-1 Compile Errors window

To see the line described in the message, double-click the message, or select it and press Return. If the error is in a source file, the THINK Editor opens the file and selects the line, as in the figure below. The THINK Editor keeps track of your edits.

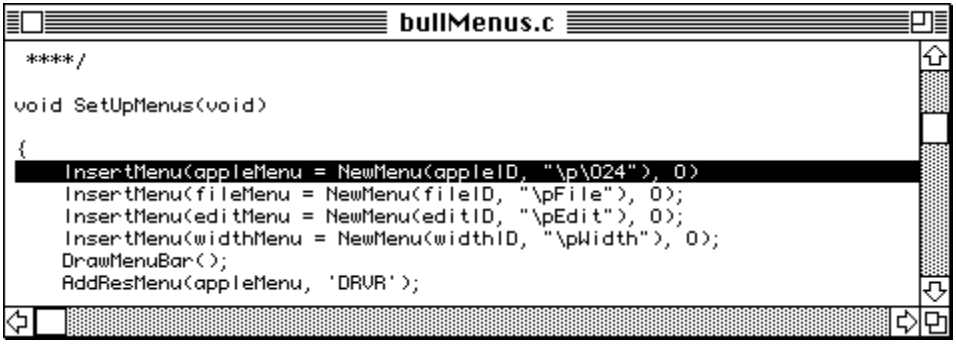


Figure C-2 THINK Editor with incorrect line of code highlighted

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 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 **Source** 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 THINK 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++ Compiler Error Messages

This list contains error messages that the compiler may generate:

'*identifier*' is a pure virtual function
The compiler cannot directly call a pure virtual function.

'*identifier*' is already defined
The object is already declared.

'*identifier*' is a virtual base class of '*identifier*'

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 not a class template

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.

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

The `struct`, `union`, or `enum` tag identifier includes invalid characters or is already defined.

'*identifier*' is not a member of struct '*identifier*'

See ARM 11.4 for more information.

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 struct or a class

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 in function parameter list

The parameter *identifier* is not listed as a parameter to the function in the function definition.

'*identifier*' is not a static variable

A static variable is not used as an argument to a static constructor when required.

'*identifier*' must be a base class

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 '::'

The identifier before the double colon operator must be a `class`, a `struct`, or a `union`.

'*identifier*' must be a public base class

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

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

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_

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

A macro has more than the allowed number of macro parameters.

':' expected

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.

'::' or '(' expected after class '*identifier*'
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
The compiler expects a semicolon at the end of a statement.

']' expected
The compiler expects a close bracket at the end of an array declaration or reference.

'(' expected
The compiler expects the expression after the `if`, `while`, or `for` keywords to be enclosed in parentheses.

')' expected
The compiler expects a set of parentheses to be closed. Check for a pair of mismatched parentheses or a bad expression.

'{' expected
The compiler expects an open brace.

'}' expected
The compiler expects a close brace.

'{' or tag identifier expected
The compiler expects a tag name or an open brace to follow the keywords `struct`, `class`, `union`, or `enum`.

'=', ';' or ',' expected
A variable is declared incorrectly. A declaration must include an equals sign, a semicolon, or a comma after the variable name.

cannot appear at beginning or end
The double-number sign operator cannot appear at the beginning or end of a list of tokens. The operator must be placed between two tokens. For example, `a ## b`.

 # must be followed by a parameter

The number sign operator must appear in front of a macro parameter. For example, `#c`.

'#else' or '#elif' found without '#if'

More `#else` or `#elif` preprocessor directives appear than preceding `#if`, `#ifdef`, or `#ifndef` directives.

'#endif' found without '#if'

More `#endif` preprocessor directives appear than preceding `#if`, `#ifdef`, or `#ifndef` directives.

'<' expected

See ARM 14.1 for more information.

In a class or function template, the argument list must be placed between angle brackets.

'>' expected

See ARM 14.1 for more information.

In a class or function template, the argument list must be placed between angle brackets.

0 expected

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

Only binary digits can follow the characters `0b`. No spaces should be placed between the `b` and the number.

access declaration must be in public or protected section

See ARM 11.3 for more information.

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
};
```

a derived class member has the same name *identifier*

See ARM 11.3 for more information.

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
};
```

alignment must be 1, 2, 4

The value for the alignment in a `#pragma align` statement must be 1, 2, or 4.

already seen initializer for '*identifier*'

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*'

This class has more than one base class, and it is not clear to which the program is referring.

ambiguous reference to function

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

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.

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.

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

A negative number or zero cannot act as an array dimension when you declare an array.

array of functions or refs is illegal

See ARM 8.4.3 for more information.

An array of pointers to functions, not an array of functions, can be declared. For example, instead of this:

```
int (&x[10])();
      // ERROR: an array of functions
      //        returning int
```

use this:

```
int (* x[10])();
      // OK: an array of pointers to
      //     functions returning int
```

array or pointer required before '['

The brackets operator can only follow an array or pointer identifier.

assignment to 'this' is obsolete, use X::operator new/delete

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&

An operator overloaded function that is not a class member must have at least one parameter that is a class or class reference.

bad member-initializer for '*identifier*'

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

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
```

blank arguments are illegal

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

The `break` statement can occur only within a `for`, `while`, `switch`, or `do`/`while` statement.

can only delete pointers

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

A new value is assigned to a `const` variable. Remove the assignment or remove the restriction from the variable.

can't declare member of another class *identifier*

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

Having a constructor as a default function parameter is illegal. For example:

```
class X {
public:
   X(int);
};

void foo( X = X(1) );// ERROR: X(1) is a
                     //        constructor.
```

can't have unnamed bit-fields in unions

Using an unnamed bit-field in a union is illegal. Use a named bit-field or remove the bit-field.

can't nest comments

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

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

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

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

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****

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*'

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

`extern` is an illegal storage class for a function parameter.

cannot delete pointer to const

See ARM 8.5.3 for more information.

Using the delete operator on a `const` pointer is illegal. Remove the `const` casting, or remove the delete operator.

cannot find constructor for class matching *name*

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
}
```

See ARM 12.1 and 12.8 for more information.

cannot generate *identifier* for class '*identifier*'

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

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*'

The constructor initializer can initialize only non-static members.

cannot implicitly convert

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.

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

This value already occurs as a case within the `switch` statement.

casts and sizeof are illegal in preprocessor expressions

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.

class name *identifier* expected after ~

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

The code contribution of one file exceeds 32K.

comma not allowed in constant expression

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

Non-extern `const` declarations or references must be initialized.

constant expression does not fit in type

Each constant expression evaluates to a constant in the range of representable values for its type.

constant initializer expected

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

A `continue` statement occurs out of context. Use it only within `for`, `while`, and `do`/`while` statements.

data or code *'identifier'* defined in precompiled header

Precompiled headers can contain only declarations, not definitions.

declarator for 0 sized bit-field

A bit-field must have a size.

'default:' is already used

The `default:` statement appears more than once in a `switch` statement.

delete[] *identifier* not allowed for handle/Pascal class

You cannot use delete on an array of Pascal or handle-based objects.

different configuration for precompiled header

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.

divide by 0

A constant expression tries to divide by zero or get modulo (%) of zero.

duplicate direct base class '*identifier*'

See ARM 10.1 for more information.

When you are declaring a new class, the same class occurs more than once in its list of direct base classes.

empty declaration

A declaration must declare at least a declarator, a tag, or the members of an enumeration.

end of file found before '#endif'

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*

A missing `*/` causes the compiler to reach the end of the file in the middle of a comment.

end of line expected

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
```

exponent expected

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

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

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

An inappropriate type occurs for a member of a bit-field structure. Use signed/unsigned `char`, `short`, `int`, or `long`.

filespec string expected

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

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 '*identifier*' has no prototype

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

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.

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

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
Anonymous unions cannot have function members.

global anonymous unions must be static
See ARM 9.5 for more information.
Anonymous unions must be `extern` or `static`.

hex digit expected
The compiler expects to find a hexadecimal digit after the characters `0x`. Do not put any white space after the `x`.

identifier expected
The compiler expects to find an identifier, but finds instead another token.

identifier found in abstract declarator
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 254 chars
The maximum size of an identifier is 254 characters.

identifier or '(declarator)' expected
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 left braces than right braces.

illegal cast
It is illegal to cast an object to an inappropriate type. For example, `struct` or `union` cannot be cast to other types.

illegal character, ascii *number* decimal
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

Certain types cannot occur together. For example, you cannot declare a variable to be a `short long int`.

illegal constuctor or destructor declaration

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 operand types

The operands are of the wrong type. Cast the operands to the correct type.

illegal parameter declaration

The parameter declaration is formed improperly. For example, an old-style declaration may not declare one of the parameter's types:

```
void f(x, y)
x;              // ERROR: Left out the type.
int y;          // OK
{
   // . . .
}
```

Another example is a declared function with a `#pragma parameter` that takes more than five arguments:

```
#pragma parameter g(a1,a2,a3,a4,a5,a6)
```

illegal pointer arithmetic

The only legal operations on pointers are adding or subtracting an integer from a pointer; subtracting a pointer from another pointer; and comparing two pointers with <, >, ==, <=, or >=.

illegal return type for operator->()

See ARM 13.4.6 for more information.

`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

Variables cannot be of type `void`.

```
struct X {
   void var;   // ERROR
}
```

inherited function must be member of derived class

When using the inherited `::`, the member being accessed must exist in the first base class of the specified object.

initializer for static member must be outside of class def

See ARM 9.4 for more information.

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

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

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*

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 reference initialization

See ARM 8.4.3 for more information.

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

Friend functions cannot be virtual.

last line in file had no \n

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

The line number in the `#line` directive must be a constant expression.

linkage specs are "C", "C++", and "Pascal", not "*identifier*"

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.

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

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
}
```

main() cannot be static or inline

See ARM 3.4 for more information.

It is illegal to declare the function `main()` as `static` or `inline`.

maximum width of *number* bits exceeded

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
};
```

macro '*identifier*' can't be #undef'd or #define'd

It is illegal to redefine or undefine this predefined macro.

See ARM 14 for more information.

malformed template declaration

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 length of *number* exceeded definition

A macro was seen that was larger than the compiler's internal buffer.

member *'identifier'* can't be same type as struct *'identifier'*

A structure cannot contain itself as a member, as in:

```
struct X {
   struct X x;
};
```

member '*identifier*' is const but there is no constructor

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

A class member that is private or protected cannot be accessed.

member '*identifier*' of class '*identifier*' is private

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

If you use the ANSI conformance option, you cannot declare a member function to be static.

must be void operator delete(void * [,size_t]);

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**)**

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*`.

new *identifier* [], not allowed for handle/Pascal class

You cannot allocate an array of Pascal or handle-based objects using `new`.

must use delete[] for arrays

See ARM 5.3.4 for more information.

To delete an array `a`, use this statement:

```
delete[] a; // OK
```

and not

```
delete a;   // ERROR
```

no constructor allowed for class '*identifier*'

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'*

A static function was declared but never defined.

```
static void f(void);
void  g(void)
{     f();
}
```

no instance of class *'identifier'*

You get this error message for attempting to reference class members in a class static function.

no identifier for declarator

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*' for member '*identifier*'

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*'

The function is overloaded and the compiler cannot find a function that matches the call.

no return value for function '*identifier*'

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 tag name for struct or enum

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.

Warning. In most cases, this message means that a temporary occurs and the warning initializes the reference to that temporary. 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.

not a struct or union type

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

You cannot overload these operators:

```
.        .*       ::       ?:       sizeof
#        ##
```

not in a switch statement

It is illegal to use a `case` or `default` statement outside a `switch` statement.

number '*number*' is too large

The number is too large to be represented in an object with `long` type.

number is not representable

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 C-1 Unrepresentable numbers

object has 0 size

The compiler does not allow objects of zero size. Trying to subtract two pointers that point to zero-sized objects causes division by zero.

octal digit expected

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*'

Member initializers in which the member lacks a constructor must have exactly one parameter because the member is initialized by assignment.

only classes and functions can be friends

It is legal to declare other classes or functions `friend` only when declaring a function within a class.

only one identifier is allowed to appear in a declaration appearing in a conditional expression

When declaring identifiers in `if`, `for`, `while`, and `switch` statements, only one identifier is allowed.

only pointers to handle based type allowed

You cannot declare an instance of a handle object or Pascal object. For example:

```
class x__Pasobj
{
}
x y;
```

operator functions ->() and [] must be non-static members

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

It is illegal to declare an overloadable operator as a variable. For example:

```
struct X {
   int operator<<;   // ERROR
};
```

out of memory

The compiler is out of memory. Try the following:

- Break the file or function into smaller units
- Increase the partition size for the THINK Project Manager
- Close any open windows in the editor

overloaded function '*identifier*' has different access levels

See ARM 11.3 for more information.

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

Pascal object classes do not allow overloaded functions or operators.

parameter list is out of context

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*'

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

You cannot use C++ virtual functions in a code resource.

pointer required before '->', '->*' or after '*'

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 ->*

The identifier after `.` or `->*` must be a pointer to a member of a `class` or `struct`.

pointers and references to references are illegal

You cannot declare a pointer or reference to reference type, as in:

```
int & & a;
```

possible extraneous ';'

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

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.

pragma parameter function prototype not found

This error occurs when a `#pragma` parameter is not followed by the function declaration.

premature end of source file

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 function.

prior forward reference class *identifier* must match handle/Pascal class type

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*

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

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.

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

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.

return type cannot be specified for conversion function

See ARM 12.3.2 for more information.

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*'

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

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

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

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

Static member functions of classes cannot be virtual.

static or non-member functions can't be const or volatile

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
It is illegal to declare a static variable within an inline function.

storage class for '*identifier*' can't be both extern and inline
It is illegal to use the inline type specifier for a function declared external.

string expected
The compiler expects to encounter a string but cannot find one. Check for an `#ident` directive not followed by a string.

template-argument '*identifier*' must be a type-argument

See ARM 14.4 for more information.

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.

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.
   // ...
}
```

too many characters in character string
The character literal has more than four characters. Try splitting the literal into two or more smaller ones. For example:

```
char a = '1234';     // OK
char b = '12345';    // ERROR: Too big
```

too many initializers
The item contains too many initializers. For example:

```
char a[3]="hello";// ERROR
char c[3]="hi"    // OK
```

trailing parameters must have initializers

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

It is illegal to declare a type conversion function outside a class. Declare it inside a class.

type is too complex

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

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(size_t [,..]);

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 must be void **operator new(Pascal void (*) ( ), size_t)

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 of '*identifier*' does not match function prototype

The arguments of the function do not match the prototype previously given.

undefined escape sequence

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 C-2 Defined escape sequences

undefined identifier '*identifier*'

It is illegal to use an identifier without declaring it. Spell the identifier correctly.

undefined label '*identifier*'

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*'

The structure or union is not defined.

undefined use of struct or union

It is illegal to use operators, such as arithmetic or comparison operators, on a `struct`, `class`, or `union`.

union members cannot have ctors or dtors

A union cannot contain a member that is an object of a class with a constructor or a destructor.

unrecognized pragma
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*'
The compiler does not support the specified preprocessor directive.

unrecognized token
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
A macro argument is missing a close quote or parenthesis.

unterminated string
A string is missing a close quote, or a file contains a lone quote mark.

use delete[] rather than delete[expr], expr ignored
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)
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
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

Warning. The optimizer discovers that a specified variable appears before it is initialized. The program may generate inexplicable results.

vectors cannot have initializers

It is illegal to initialize a vector of objects with a constructor that has an argument list.

very large automatic

Warning. Large automatic variables can cause stack overflow. Dynamically allocate the memory with a function such as `malloc()`.

voids have no value, ctors and dtors have no return value

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

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
```

Index

Entries in **boldface** are menu commands. Entries in `typewriter face` are functions, methods, variables, keywords, or files.

Numeric/Symbol

A

B

C

D

E

F

G

H

N

O

T

U

V

W

Symantec C++
Disk Exchange and/or Replacement

DISK EXCHANGE: Symantec C++ is available on high density (1.5MB) and double-sided (800KB) disks. If you have purchased a product that does not contain the correct disk size for your computer, you may exchange the disks at no extra charge. Simply fill out Section A, enclose all original disks, and mail to the address below.

DISK REPLACEMENT: After your 60-Day Limited Warranty, if your disk(s) becomes unusable, fill out Sections A & B and return 1) this form, 2) your damaged disks, and 3) your payment (see pricing below, add sales tax if applicable), to the address below to receive replacement disks. *DURING THE 60-DAY LIMITED WARRANTY PERIOD, THIS SERVICE IS FREE.* You must be a registered customer in order to receive disk replacements.

SECTION A – FOR DISK EXCHANGE *AND* REPLACEMENT

Please Send Me: ❑ High Density Disks (1.5MB) ❑ Double-sided Disks (800KB)

Name _____

Company Name _____

Street Address *(No P.O. Boxes, Please)* _____

City _____ State _____ Zip/Postal Code _____

Country* _____ Daytime Phone _____

Software Purchase Date _____ Version _____

*This offer limited to U.S. and Canada. Outside North America, contact your local Symantec office or distributor.

SECTION B – FOR DISK REPLACEMENT ONLY

Briefly Describe the Problem: _____

_____

Disk Replacement Price	$10.00
Sales Tax (See Table)	_____
Shipping & Handling	$ 8.00
TOTAL DUE	$ _____

SALES TAX TABLE: AZ (5%), **CA** (7.25%), **CO** (3%), **CT** (6%), **DC** (6%), **FL** (6%), **GA** (4%), **IL** (6.25%), **IN** (5%), **IA** (5%), **LA** (4%), **ME** (6%), **MA** (5%), **MD** (5%), **MI** (4%), **MN** (6.5%), **MO** (7.725%), **NC** (6%), **NJ** (6%), **NY** (4%), **OH** (5%),**PA** (6%), **SC** (5%), **TX** (6.25%), **VA** (4.5%), **WA** (6.5%), **WI** (5%), **Canada** (7% G.S.T.). Please add local sales tax (as well as state sales tax) in AZ, CA, GA, LA, MN, NC, NY, OH, SC, TX, WA, WI, VA.

FORM OF PAYMENT** (Check One)

❑ Check (Payable to Symantec) Amount Enclosed $_____ ❑ Visa ❑ MasterCard ❑ American Express

Credit Card Number _____ Expires _____

Name on Card *(Please Print)* _____ Signature _____

**U.S. Dollars. Payment must be made in U.S. dollars drawn on a U.S. bank.

MAIL YOUR DISK EXCHANGE AND/OR DISK REPLACEMENT ORDER TO:

Symantec Corporation
Attention: Disk Replacement/Exchange
P.O BOX 10849
Eugene, OR 97440-2849

Please allow 2-3 weeks for delivery.

SYMANTEC.